Efficient Programming Model for OpenMP on Cluster-Based Many-Core System

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von

M.Sc. Hayder Hekmat Sulman Al-Khalissi
geboren am 12.04.1983
in Bagdad

Eingereicht am: 06.08.2015
Disputation am: 24.09.2015
Referent: Prof. Dr. Ing. Mladen Berekovic
Referent: Prof. Dr. Ing. Harald Michalik

(2015)

"The most complete gift of God is a life based on knowledge."

Imam Ali ibn Abi Talib

Abstract

TODAY'S trends show an increase in the number of cores integrated onto a single chip. There are systems with 100 or even more cores on the market – and the core numbers have been increasing steadily, laying the basis of today's many-core era (in contrast to the former multi-core era). To ensure system scalability, several clusters have been interconnected through a network-on-chip (NoC), which, of course, introduces non-uniform memory access (NUMA) effects.

As the complexity of such *systems-on-chip* (SoCs) continues to increase, it is no longer possible to ignore the challenges caused by the convergence of software and hardware development [1]. This involves attempts to deal with the hierarchical design – in which several cores are grouped in clusters or tiles – to ensure low-latency, high-bandwidth local communication by relying on fast local memories. From a programmer's perspective, it is desirable to make use of these peculiarities of the hardware, which must be clearly and carefully taken into account when designing the support for high-level parallel programming models.

This dissertation overcomes many scalability bottlenecks in cluster-based many-core systems and introduces the OpenMP programming model as a means of simplifying application development. OpenMP represents an abstraction of the programmer's view by providing abundant directives that decompose loops in sequential programs and lead to parallel programs. Further, it provides help in dealing with the segmented memory space.

In this work, the full OpenMP model is implemented on a specific instance of a cluster-based many-core system: the Intel Single-chip Cloud Computer (SCC). In this thesis, a lightweight and highly optimized runtime layer for OpenMP execution and memory model by generating the parallel code that is automatically compiled by native back-end compiler (GCC 4.6) that linked with the runtime library. I argue the case that the OpenMP model is a particularly appropriate programming model for today's – and, most probably, also for tomorrow's – many-core systems.

In this dissertation, I will address an efficient design approach of the OpenMP programming model for the Intel SCC as an example for cluster-based systems. All experimental results are evaluated with the GCC compiler and a custom implementation of the runtime library. The SCC OpenMP runtime library is designed to cope with three main challenges in a non-cache coherent system:

1. Executing unmodified legacy OpenMP programs on such system.

2. Landing OpenMP memory model on the SCC.

3. Synchronization in the work of parallel threads accounts for a sizeable fraction of an application's execution time. Therefore, an efficient implementation of the underlying synchronization algorithms and their underlying synchronization primitives is required, allowing high-level barrier constructs to deliver a good performance.

This thesis shows that architectural awareness is the key to support efficient and streamlined fork/join primitives. Therefore, hierarchical fork/join operations are compared to "flat" ones, where there is no notion of the hierarchical interconnection system. Next, a new extension for the OpenMP compiler is developed to improve the application performance. This extension is a new directive that is used to keep the shared data in local memory (L2 cache) to be used again by the next parallel region in the same thread. Based on this directive, the compiler can skip the flush routine at the end of the parallel region, which is used to ensure that the data in the global shared memory is updated. Thus, an average 1.20x speedup in the LU-decomposition benchmark is achieved, in comparison to the already implemented OpenMP scheme.

Furthermore, the effectiveness of OpenMP is demonstrated on a set of widely used kernels and real-world applications. An extensive set of experiments shows how this model achieves significant parallel speedups up to 48x in several applications.

List of my Publications

- H. Al-Khalissi und M. Berekovic: *Performance of RCCE broadcast algorithm in SCC* MARC Symposium, Karlsruhe Institute of Technology Aachen, 2011
- H. Al-Khalissi, A. Marongiu und M. Berekovic: *Low-overhead Barrier Synchronization for OpenMP-like parallelism on the Single-chip Cloud Computer*MARC Symposium. RWTH Aachen University, 2012
- H. Al-Khalissi, A. Marongiu und M. Berekovic: *An approach for Supporting OpenMP on the Intel SCC* SPLASH-MARC Symposium, 2013 (Best award)
- H. Al-Khalissi, R. Buchty und M. Berekovic: *Efficient Barrier Synchronization for OpenMP-like parallelism on the Intel SCC* ICPADS, 2013
- H. Al-Khalissi, S. A. A. Shah und M. Berekovic: *An efficient Barrier Implementation for OpenMP-like parallelism on the Intel SCC* PDP, 2014
- H. Al-Khalissi, A. Marongiu und M. Berekovic: *On the Relevance of Architectural Awareness for Efficient Fork/Join Support on Cluster-Based Manycores* Proceedings of International Workshop on Manycore Embedded Systems. USA, 2014

Acknowledgements

Over the years of the journey for research which culminated in this dissertation, many people have given me more of their time, companionship, professional and personal help. These years have been filled with interactions with people that enriched my life, both personally and academically. Above all, patience was a key to finishing this dissertation by my seeming determination to indefinite position the deadline. It is with heartfelt appreciation, I am glad to express my gratitude to who accompanied me on this thesis journey.

I would first of all like to say a sincere danke schoen to my advisor, Professor Mladen Berekovic for his fundamental role in my doctoral work. He gave me the scientific support and supervision that any graduate student can expect from his professor. I remember the first time when I met with him, he told me; "I am not only here as a supervisor, you can consider me as a father". He strived to create a pleasurable atmosphere in my research and always encouraged me to continually work on my thesis.

I am also very grateful to Dr. Rainer Buchty, with whom I had many very fruitful discussions on various topics of this thesis. I also wish to thank Dr. Andrea Marongiu (PostDoc researcher at IIS Lab - ETH Zurich - PostDoc researcher at DEI - University of Bologna) very much for all his contributions and input, he helped me with OpenMP compiler extension, in particular extending the GCC compiler to translate OpenMP shared data to runtime support code. Grazie Andrea, for our productive collaboration and, especially, for the fact that you always reserved plenty of time for me when I visited or contacted you – both, time during the days and the evenings.

Thanks to my good friend Syed Abbas Ali Shah with whom I shared the ups and downs, typical of a doctoral path. Thanks to my friends Shaodong Qin, Bastian Farkas, Ioanna Teskoura, Rolf Meyer, Thomas Schuster, Jamin Naghmouchi, and Sven Horsinka for making me feel like at home while being thousands of kilometers away from it.

In this context, I am grateful to the German Academic Exchange Service (DAAD) for its financial support that made all these contacts would have been possible. The DAAD's scholarship is highly appreciated as the key to open the door for my Doctoral Study in Germany. I wish to express my sincere gratitude to Intel for providing SCC system and software development platform. In line with the acknowledgement to Intel, I owe sincere thankfulness to Mr. Mark Aughenbaugh and Mrs. Julie Donnelly for their prompt and kind support.

Moving towards more personal acknowledgements, I would like to execute a big fork() of aggregated thanks towards all my sisters, brothers, and friends for their help and patience, and for the fact that they never gave in to the temptation to make fun of my perennial thesis delays and woes.

I am, of course, particularly indebted to my parents who gave me unconditional support to achieve this mile stone in my life – they made me what I am today. I thank them for their monumental, unwavering support and encouragement on all fronts. Especially for my mother (أمي), without her prayer and tears in her eyes (not seeing me for 4 years) none of this would have been even remotely possible

Finally, I thank my God for all of you, and many thanks for all the memories.

Contents

List of Figures

List of Tables

Abbreviations

AIC	**A**tomic **I**ncrement **C**ounter
API	**A**pplication **P**rogramming **I**nterface
APIC	**A**dvanced **P**rogrammable **I**nterrupt **C**ontroller
ASO	**A**verage **S**lave **O**verhead
BIOS	**B**asic **I**nput/**O**utput **S**ystem
BT-LUTB	**B**inary-**T**ree-**L**ook-**u**p **T**able **B**arrier
BMC	**B**oard **M**anagement **M**icro-controller
BT-PB	**B**inary-**T**ree **P**olarity **B**arrier
CFG	**C**ontrol **F**low **G**graph
CL1INVMB	**C**ache **L**ine**1** **INV**alid **M**essage **B**uffer
C-LUTB	**C**hain-**L**ook-**u**p **T**able **B**arrier
CPB	**C**hain-**P**olarity **B**arrier
CPB-UC	**C**hain-**P**olarity **B**arrier-**U**n-**C**ached
CR4	**C**ontrol **R**egister **4**
CRF	**C**ommit-**R**econcile and **F**ence
DBT-PB	**D**ouble **B**inary-**T**ree **P**olarity **B**arrier
DDR-SDRAM	**D**ouble **D**ata **R**ate-**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory
DIMM	**D**ual **I**n-line **M**emory **M**odule
DIV	**Div**ergence
DMA	**D**irect **M**emory **A**ccess
DSM	**D**istributd **S**hared **M**emory
DTC	**D**ynamic **T**hread **C**reation
DTDs	**D**istributed **T**ag **D**irectorie**s**
DVFS	**D**ynamic **V**oltage and **F**requency **S**caling
ELF	**E**xtensible **L**inking **F**ormat

EPCC	**E**dinburgh **P**arallel **C**omputing **C**entre
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FTP	**F**ixed **T**hread **P**ool
GA	**G**lobal **A**rray
GCC	**G**NU **C**ompiler **C**ollection
GCU	**G**lobal **C**lock **U**nit
GIR	**G**lobal **I**nterrupt **R**egister
GOMP	**G**NU **O**pen**MP**
GOT	**G**lobal **O**ffset **T**able
GPU	**G**raphics **P**rocessing **U**nit
GPGPU	**G**eneral-**P**urpose **G**raphical **P**rocessing **U**nits
GTC	**G**lobal **T**ime-stamp **C**ounter
HPC	**H**igh-**P**erformance **C**omputing
HPF	**H**igh-**P**erformance **F**ortran
ICOV	**I**nstantaneous **C**oefficient **O**f **V**ariation
ID	**I**dentity **D**ocument
ILP	**I**nstruction-**L**evel **P**arallelism
INVD	**INV**alid **D**ata-L1
IP	**I**nternet **P**rotocol
IR	**I**ntermediate **R**epresentation
ITRS	**I**nternational **T**echnology **R**oadmap for **S**emiconductors
L2CFG0/1	**L2 C**ach con**F**i**G**uration **0/1**
LCL_THR_IDS	**LoCaL THR**ead **IDS**
libGOMP	library **G**NU **O**pen**MP**
LINT0/1	**L**ocal **INT**errupt **0/1**
LogP	**L**atency **O**verhead **g**ap **P**arallel-computation
LUT	**L**ook-**u**p **T**able
LUTB	**L**ook-**u**p **T**able **B**arrier
MC	**M**emory **C**ontroller
MCAPT	**M**ulticore **C**ommunication **A**pplication **P**rogramming **I**nterface
MCPC	**M**anagement **C**ontrol **P**ersonal **C**omputer
MESH	**M**emory **E**fficient **SH**aring
MIC	**M**any **I**ntegrated **C**ore

MIU	**M**esh **I**nterface **U**nit
MO	**M**aster **O**verhead
MPB	**M**essage **P**assing **B**uffer
MPBT	**M**essage **P**assing **B**uffer **T**ype
MPI	**M**essage **P**assing **I**nterface
MPSoC	**M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
M-SSB	**M**aster **S**haring-**S**lave **B**arrier
M-PSB	**M**aster **P**olarity-**S**lave **B**arrier
NFLAGS	**N**otify **FLAGS**
NoC	**N**etwork-**o**n-**C**hip
NPB	**N**AS **P**arallel **B**enchmarks
NUMA	**N**on-**U**niform **M**emory **A**ccesses
omp_data	**O**pen**MP data**-sharing structure
omp_fn	**O**pen**MP f**unctio**n**
OpenCL	**Open C**omputing **L**anguage
OpenMP	**Open M**ulti-**P**rocessing
OS	**O**perating **S**ystem
PCD	**P**age **C**ache **D**isable
PCI-e	**P**eripheral **C**omponent **I**nterconnect-**e**xpress
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
PIT	**P**rogrammable **I**nterrupt **T**imer
POP-SHM	**P**rivately **O**wned **P**ublic-**SH**ared **M**emory
POSIX	**P**orable **O**perating **S**ystem **I**nterface
PSE	**P**age **S**ize **E**xtension
PST_THR_IDS	**P**re**S**isten**T THR**ead **IDS**
PU	**P**rocessing **U**nits
PVM	**P**arallel **V**irtual **M**achine
PWT	**P**age **W**rite-**T**hrough
RC	**R**elease **C**onsistency
RCCE	**R**ocky **C**hip **C**ommunication **E**nvironment
RCCE-B	**R**ocky **C**hip **C**ommunication **E**nvironment-**B**arrier
RCKMPI	**R**o**ck**y **M**essage **P**assing **I**nterface
RDTSC	**R**ea**d T**ime **S**tamp **C**ountr

RFLAGS	**R**elease **FLAGS**
ROI	**R**egion **O**f **I**nterest
SC	**S**equenial **C**onsistency
SCC	**S**ingle-chip **C**loud **C**omputer
SccKit	**S**ingle-chip **C**loud **C**omputer **Kit**ool
shm_malloc	**sh**ared **m**emory **malloc**
SIF	**S**ystem **I**nter**F**ace
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
S-L2	**S**hared-**L2**-cache
SMC	**S**oftware **M**anaged **C**oherency
SMP	**S**ymmetric **M**ulti-**P**rocessors
S-MPB	**S**hared-**MPB**
S-MSB	**S**hared-**M**aster-**S**lave **B**arrier
S-MSBO	**S**hared-**M**aster-**S**lave **B**arrier-**O**ptimized
S-MSBO-UC	**S**hared-**M**aster-**S**lave **B**arrier-**O**ptimized-**U**n-**C**ached
SoC	**S**ystem **o**n **C**hip
S-Off	**S**hared-**Off**-chip
SPEC	**S**tandard **P**erformance **E**valuation **C**orporation
SRAD	**S**peckle **R**educing **A**nisotropic **D**iffusion
SRAM	**S**tatic **R**andom-**A**ccess **Memory**
SVP	**S**elf-adapive **V**irual **P**rocessor
TACO	**T**opoloies **A**nd **CO**llections
TCDM	**T**ightly-**C**oupled **D**ata **M**emory
TCP	**T**ransmission **C**ontrol **P**rotocol
TEM_DESC_PTR	**TE**a**M** **DESC**riptor **P**oin**T**e**R**
TFLOPS	**T**era **FL**oating-point **O**perations **P**er **S**econd
TLP	**T**hread **L**evel **P**arallelism
T&S	**T**est-**&**-**S**et
UC	**U**n-**C**ached
UMA	**U**niform **M**emory **A**ccesses
UPC	**U**nified **P**arallel **C**
VPU	**V**ector **P**rocessing **U**nit
VRC	**V**oltage **R**egulator **C**ontroller

vSMP	**v**ersatile **S**ymmetric **M**ulti-**P**rocessors
WB	**W**rite-**B**ack
WBINVD	**W**rite-**B**ack **INV**alid **D**ata
WCB	**W**rite **C**ombine **B**uffer
WT	**W**rite-**T**hrough
XMP	e**X**tension for **S**calable and performance-aware **P**arallel programming

Dedicated to my parents and my siblings

Chapter 1

Introduction

1.1 The Realm of Many-Core

According to the Moore Law in the semiconductor industry – which was published by Gordon Moore, founder of Intel Corporation, in 1965 – the number of transistors on a chip will roughly double every two years [6]. Accordingly, this law implies a picture of the future that is full of best high-technology products, fast and cheap [7]. As a consequence, this empirical law describes the increasing density of transistors permitted by technological advances and also imposes a new set of requirements and challenges. The clock speeds of microprocessors keep increasing, exploiting instruction-level parallelism (*ILP*) through pipelining out-of-order execution and super-scalar issue [8, 9]. Clearly, the system performance has increased and the costs are reducing, but, unfortunately, Moore's Law has started to reveal problematic aspects in this field [10], because of the power density and ILP limits.

However, the transparent acceleration cannot be pushed any further due to ILP, and making the cores more complex won't help. Due to these obstacles, a new phase of development had to be initiated, resulting in the birth of the multi-core technology at the beginning of the 21st century. There was a shift towards real parallelism by integrating a number of processors on a single computing component [11]. Hence, we now have multi-core processors (i.e. addressing thread- and task-level parallelism), not so much because of the energy density (which would also affect the entire chip), but for the simple reason that we cannot get more benefits from applying even more ILP, DLP, and speculation techniques. All we can use Moore's law for now is to exploit TLP – which in term bears the problem of interconnect and programmability (*explicitly parallel programming*). We are currently facing a jumble of measures, from hardware-aware programming to low-level parallel models (Message Passing Interface (MPI)) to some nicer, but nevertheless

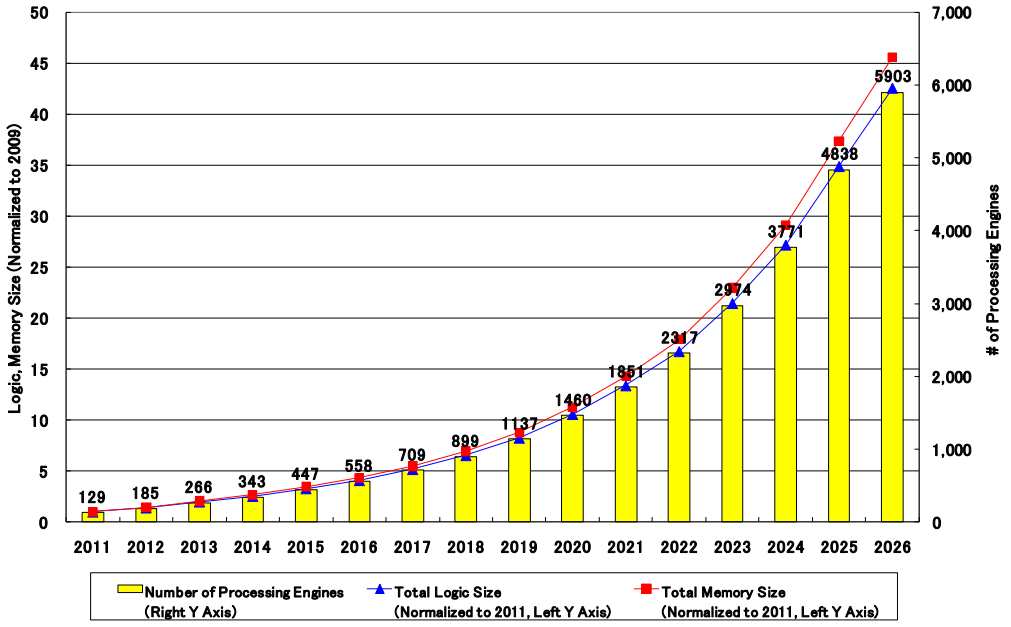


FIGURE 1.1: Predicted processor number for SoC consumer portable design [2]

explicitly formulated approaches (CUDA, High-performance Fortran) to shared memory (Open Multi-Processing (OpenMP)).

Figure 1.1 shows the *International Technology Roadmap for Semiconductors* (ITRS) 2011 predication for the number of processing engines and memory size to be integrated in future system-on-chip (SoC) devices [2]. According to the ITRS, future platforms will show a continuous increase in the number of processing cores in SoC, by a factor of 3.5 times every five years. Consequently, the ITRS figure illustrates the increasing complexity due to multiplying the number of processing cores. Furthermore, the memory size and logic size will follow the same trends. In this context, there are many prominent examples (such as Intel's Single-chip Cloud Computer (SCC) [4] with 48 cores, Intel Xeon Phi [12] with more than 240 cores, and Tilera's with 100 cores [13]) showing that the semiconductor integration is not the main issue. Today, a *Multi-processor System–on–Chip* (MPSoC) is an SoC that holds one or more types of computing units, memories, input/output devices, and other peripherals. Potentially, MPSoC are classified as *many-core* to express a high core count. Its architecture contains at least ten loosely coupled (possibly heterogeneous) simpler processors. This, in consequence enables a more power-efficient parallel approach instead of relying on higher speeds and, thus, higher power consumption [14]. The cores are structured in such a way that the memory resembles a non-uniform memory architecture (NUMA) approach with (usually) incoherent caches, and every core runs its own (instance of an) OS or operates under the *Symmetric Multi-Processor* (SMP) approach.

However, there is an underlying problem to the complexity wall of the evolution of SoC platforms: the **scalability**. In parallel computing, scalability means the capability of a system to increase the total performance power when more resources are added. Arguable, the increased complexity in SoC – a factor contributing to the long-awaited breakthrough towards mainstream – can be attributed to the lack of programming tools. Another factor is that, over the years, the parallel hardware was continuously improved (in terms of power, price, availability, etc.). As parallel software has always lagged behind and failed to meet expectations, it was difficult to actually use the hardware in a profitable manner. Sometimes, apparently, radically different parallel systems are designed even before the old ones could by programmed and used properly.

In this thesis, the terms 'core' and 'processor' are used as synonyms, despite their physical differences.

1.2 Motivation

As discussed above, the number of cores and the complexity of a single node keep increasing. Nowadays, the range of available software has to ensure that the systems' tremendous peak performance can be used efficiently. This, of course, requires to employ all the available cores for most of the time, as a way to potentially execute more instructions simultaneously and to maximize the performance. In MPSoC architectures, many concessions are made regarding the programmability by changing several aspects of the architecture in favour of hardware scalability, reduced complexity of the design, production costs, or energy efficiency. These changes to the hardware are reflected in the programming model, and consequently impose a new set of challenges on the programmer. Furthermore, knowledge and experience in parallel system programming have not kept pace with the trend towards parallel hardware, which will result in meagre performance. In consequence, programmers have to be able to cope with parallel computation and to manage the memory hierarchy in such systems — adding more complexity to the software and posing a quite a challenge for most programmers. Due to the requirements of parallel system development, producing software has become a tedious task, especially considering the revolution in the field of hardware design, which leads to newly introduced platforms every month. Despite the extensive collection of parallel programming models, the software communities are still lagging behind and are not yet coping with the parallelism requirements posed by the new generation of hardware. As a consequence, the application developers are now confronted with parallel programming techniques to solve the problems that require large resources in order to achieve high performance.

Unfortunately, achieving high performance is not the only challenge in the field of parallel computing. Any approach towards parallel programming has to overcome many challenges such as productivity, usability, and portability. One of the main challenges the parallel computing industry has to tackle is the task of bridging the gaps between the hardware and software to increase the capabilities of parallel system computing [15]. Naturally, there is a vast range of applications from various fields that has to be adapted to function in a plethora of parallel environments. A common approach would be to try and find a solution to the problem with minimal *effort* and in minimum *time* [15]. To ensure the functionality of a parallel application based on the development time, the programmer needs support to develop an application under a parallel programming model.

The key is to define programming models in such a way that the complexity of the trends mentioned above is hidden from the programmer. Here, the compiler and a run-time system have all necessary information to deal with low-level hardware privileges automatically, efficiently, and correctly. Using the shared memory paradigm and employing high-level parallel programming abstractions such as OpenMP [16], it is possible to ease the efforts based on this paradigm. The distributed memory (e.g, MPI) needs an explicit communication layer between all rounds, which requires many access points to the main memory, whereas shared memory does not require an explicit copy of data.

OpenMP is a system-independent set of procedures and software that aims to provide high-level parallel language that support a wide range of applications – from automotive and aeronautical to biotech, automation, robotics and financial analysis. Recently, the simplification of use has led to a flourishing number of OpenMP implementations for MPSoCs. In order to make it accessible to traditional sequential programmers, since in fact the OpenMP provides high-level abstractions to increase program development without altering the base programming language. Many architectural templates were exploited for implementing OpenMP, each dealing with specific hardware features. Naturally, customizing an implementation for a target platform will enable high performance on that machine.

This thesis addresses the problem of a full-design OpenMP regarding a cluster-based many-core single chip that typically features complex memory systems, with explicitly managed SRAM banks and NUMA organization. Intel's Single-chip Cloud Computer (SCC) [4] is a good example in this respect, and it poses several challenges to accommodate the OpenMP execution model. First, each core runs a separate instance of the operating system, which makes it impossible to run existing OpenMP implementations based on a standard threading library (e.g., Pthreads) directly. Second, barrier primitives should leverage fast and local memories (i.e, scratchpad) to minimize the time of

the inter-thread synchronization. Third, data sharing is not at all trivial, as OpenMP assumes a flat memory model, which is unmatched by the distinct private virtual memory segments seen by different SCC cores. Furthermore, the OpenMP execution model assumes the system to be a homogeneous resource (processors and memories) with a physically shared memory (e.g, symmetric multiprocessor) when partitioning the workload among available threads. Moreover, NUMA memory breaks this assumption, since accessing shared data will result in different latencies from different threads.

It is believed that extending an OpenMP model (by adding more directives) will not prove that the model is scalable with the number of cores added. On the contrary, this solution will add more complexity to the attractive programming model which is characterized as easy and simple in its usage. Currently, OpenMP is targeting heterogeneous systems with a huge number of directives to tackle the hardware complexity, and the further development will continue to improve the functionality of such systems.

1.3 Thesis Contributions

The ultimate goal of this dissertation is to design and improve the scalability of OpenMP performance on many-core architectures. Therefore, this thesis presents several novel approaches to adapt OpenMP to cluster-based many-core platforms.

Firstly, in order to estimate the potential for performance improvements, the state-of-the-art techniques have to be investigated, addressing the implementation challenges to support the OpenMP execution model on top of such platforms.

To avoid limiting effects regarding the parallelization effectiveness, the optimized low-level library-based API is designed from scratch so as to efficiently apply the parallel character of OpenMP applications to the SCC platform.

Further, the GCC 4.6 compiler is customized and extended to support code transformations and instrument access to shared data in the program with address translation routines on the MPSoC system. GCC was chosen as a starting point because it is widely used and a robust, open source implementation of the OpenMP translation pass and runtime library (**libGOMP** [17]).

The largest possible number of implementations of OpenMP-like barrier algorithms were considered in order to determine which of them is the most suitable implementation based on the underlying hardware architecture and number of threads. To find out what overhead is associated to the implementation of barrier phases, a new methodology is

proposed to classify the overhead into two sites, Master and Slaves, trying to cover the entire time consumption.

NUMA access on MPSoCs requires specific support that has significant associated overheads. As the overheads may exceed the benefits of parallelism when there is only a small amount of parallel work, the efficient implementation of OpenMP plays an important role in order to minimize thread overheads, to optimize memory access and communication as well as possible. The novel approach in this thesis aims to significantly reduce the OpenMP overheads by adopting a hierarchical approach to creating and synchronizing thread teams. Furthermore, the benefits of exploiting the memory hierarchy are explored so as to achieve high performance.

An important feature for future many-core chip architectures is the development of a shared memory paradigm with a memory consistency model that is effective for small local memory sizes in each core, scalable for a large number of cores, and easy to use. The problem here is that the applications are usually not able to reach the expected performance. Here, cache utilization is one of the critical reasons. Because the data stored in the cache can often not be reused, applications still suffer from large amounts of cache traffic (miss or flush) and the resulting long access time. This issue is especially critical for OpenMP applications on MPSoC systems since OpenMP exploits fine-grained parallelism, which leads to more data transfers between the cores, and the hardware doesn't support caching coherence. Therefore, the OpenMP implementation is supported by enabling the L2 cache for shared data and extending the compiler to handle the flush data depending on the hint from the programmer. A general idea is to avoid the overhead and the traffic which is caused by cache flush operation when the cached data is used again by the same thread. A new extension to the OpenMP model is proposed, called the *noflush* directive. Here, when the OpenMP compiler encounters the parallel region with the *noflush* clause, the compiler will disable the flush routine at the end of the parallel region. As a result, this will eliminate all the overhead of moving the data from/to the cache to/from the memory.

Finally, when the full OpenMP implementation is completed by adding the reduction clause, the performance gains are demonstrated by a sufficient number of benchmarks and applications. The implementation of OpenMP is imposed as a best programming model for environments featuring a runtime threading system that is capable of leveraging hardware primitives and symmetric shared memory system.

1.4 Synopsis and Thesis Overview

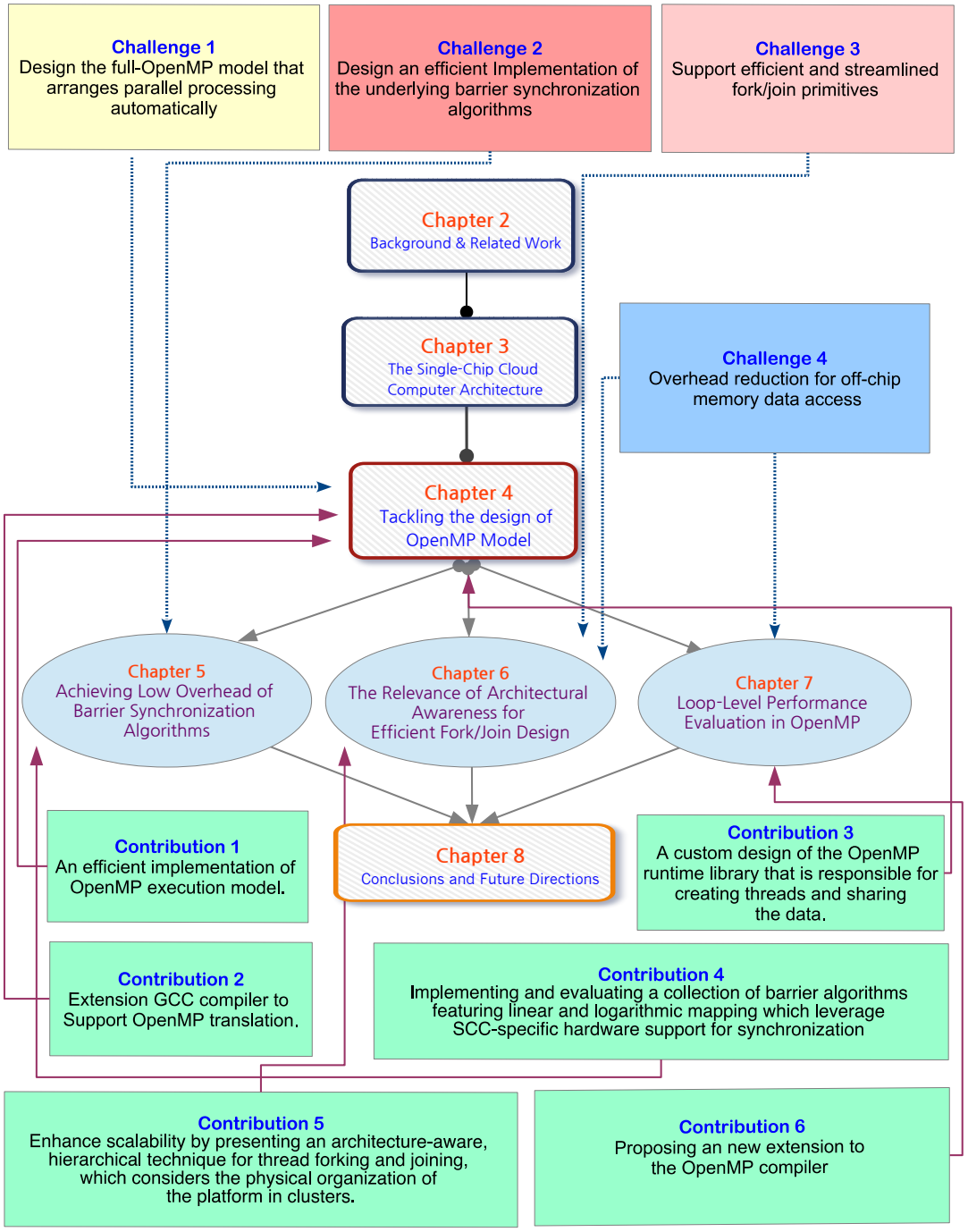


FIGURE 1.2: Fork/join model to illustrate the different chapters and the respective challenges and contributions in the scope of this thesis

The remainder of this dissertation is structured as shown in Figure 1.2:

- **Chapter 2** introduces the state-of-the-art in many-core processors, the programming languages and parallelism paradigms. It features an overview of the compiler infrastructure (GCC 4.6) upon which this work is based. Further, it focuses on

the OpenMP mapping techniques in MPSoC architectures and on a discussion of related work.

- **Chapter 3** illustrates the specification of the target system (The Single-chip Cloud Computer) in more detail – followed by an overview of the programming capabilities and a review of the programming model for the SCC platform.

  In the end, all the experimental results in this thesis are generated based on the settings listed in Section 3.9.

- The OpenMP compiler generates an optimized parallelism code for the specific platform, depending on how the programmer configured the application by adding a directive.Here, the main challenge is figuring out how to design the OpenMP compiler/translator for the MPSoC platform (i.e SCC)?. In (Chapter 4), there will be a stronger focus on each of the challenges of retargeting the OpenMP model with the features of the SCC system. Furthermore, the implementation of the OpenMP runtime environment (**libgomp_scc**) is presented,including an implicit address translation. The special extensions to OpenMP are specifically designed to take advantage of the new features of this parallel chip architecture. Methodology and micro-benchmark implementations to evaluate the parallel execution model and barrier performance are discussed in Section 4.2, while Section 4.3 focuses on some experimental results for the analysis of the runtime overhead(fork/join model and synchronization primitives).

- **Chapter 5**: OpenMP (as well as most related shared-memory-based programming models) relies on a fork/join execution model, which leverages a barrier construct to synchronize parallel threads. Barriers – implicit or explicit – are central constructs to the OpenMP execution model and to any shared memory parallel program. Therefore, the important cause for performance degradation regarding parallel program execution is the unavoidable synchronization overhead. The scalability of the implementations becomes increasingly important due to the steadily increasing number of threads for parallel regions and the complexity of memory hierarchies in the system.

  To overcome those obstacles, several barrier variants are customized to be integrated into the OpenMP runtime library. Second, a number of techniques are investigated which serve to reduce the barrier overhead by leveraging SCC-specific hardware support for synchronization and its explicitly-managed portion of the memory hierarchy (i.e., MPB), accompanied by a communication pattern analysis

similar to the barrier for message-passing machines. This is followed by a performance analysis of the performance based on the new methodology and micro-benchmarks to track a number of important methodological challenges, showing the benefits and drawbacks of the individual approaches as well as significant performance improvements connected to the best suited solutions.

- **Chapter 6**: First, many optimization techniques are surveyed for traditional (flat) fork/join implementations to reduce the overhead of forking and joining threads. Next, this chapter introduces the details of implementing a solution for the scalability bottlenecks of fork/join models in the many-core system. The solution is to adopt a hierarchical approach and, thus, to create and synchronize thread teams. The proposed solution considers the number of clusters (tiles) as the main parameters,in addition to the number of cores within each cluster. First, in the scope of thread recruiting in nesting mode, an outermost team is created with as many threads as clusters. Then, each of these threads is involved in the creation of local thread teams within each cluster. Multiple inner teams are created in parallel over different clusters, thus reducing the overall fork (join) latency. In addition, a new micro-benchmark introduced in this chapter serves to validate the performance of the fork/join mode in the flat implementation and the hierarchy implementation.

- **Chapter 7**: In Chapter 7, the performance and the effectiveness of the OpenMP model are reported by studying a set of widely used real-world applications from different problem domains. First, this chapter provides a discussion about efficiently mapping loop-level parallelism in the OpenMP model and common mistakes in measuring the performance. Then, the new extension approach is introduced for the OpenMP compiler that extends the OpenMP consistency model. This will be complemented with details about the reduction design technique in the OpenMP model used in the application. The chapter further discusses the performance results of the generated code along with the performance tuning efforts for commonly used applications after presenting the backgrounds of each application.

- **Chapter 8**: Finally, Chapter 8 contains the conclusions from my work in this dissertation as well as suggestions for future work in this context.

- **Appendix A**: This appendix describes the Intel Xeon Phi architecture with typical programming models.

- **Appendix B**: Here, additional results are presented in the scope of the Stream benchmark, based on the case-study presented in Section 7.4 and on different frequency-scalings of the SCC platform.

- **Appendix C**: This appendix contains a diagram of the history of OpenMP specifications.

Chapter 2

Background and Related Work

2.1 The landscape of Many-core Computing

MPSoC technology has entered the many-core era, with hundreds of simple processing units (PU) integrated on a single chip [18–20]. MPSoC systems have two types of architectures: *homogeneous* (SMP) and *heterogeneous* such as Cell BE processors [21] and GPGPUS [22]. Several recent many-core systems are architected as fabrics of tightly-coupled *clusters* or *tiles*. Within each cluster a small-medium number of PUs share a low-latency, high bandwidth interconnect and local memory. Scaling to hundreds of cores is achieved by replicating clusters and interconnecting them via a scalable communication medium such as a network-on-chip (*NoC*). These systems often leverage a shared memory model, where each cluster can access local or remote L1 storage, as well as L2 or L3 memories. However, due to the hierarchical nature of the interconnection system, memory operations are subject to non-uniform accesses (*NUMA*), depending on the physical path that corresponding transactions traverse. Several examples of a similar template exist: Kalray MPPA 256 [23], Tilera [13], STMicroelectronics STHORM [24], Adapteva Parallella [25], Intel's experimental 80-tile [26], Intel Xeon Phi (*MIC*) [12], Intel's prototype Single-chip Cloud Computer (*SCC*) [4], Cell Broadband Engine [21], to name a few.

Figure 2.1 shows a block diagram of the many-core template considered in this thesis. It consists of a hierarchical design, where a top-level communication system (typically a NoC, structured as a 2D mesh) interconnects a number of clusters (also known as *tiles*). Each cluster contains one or more simple cores, which can independently run an entire operating system, sharing multi-banked memory, typically implemented as SRAM blocks, and communicating via a fast local interconnection (e.g., a crossbar, or mesh of trees [24, 27]). At the top level of the various clusters shared local memory, also

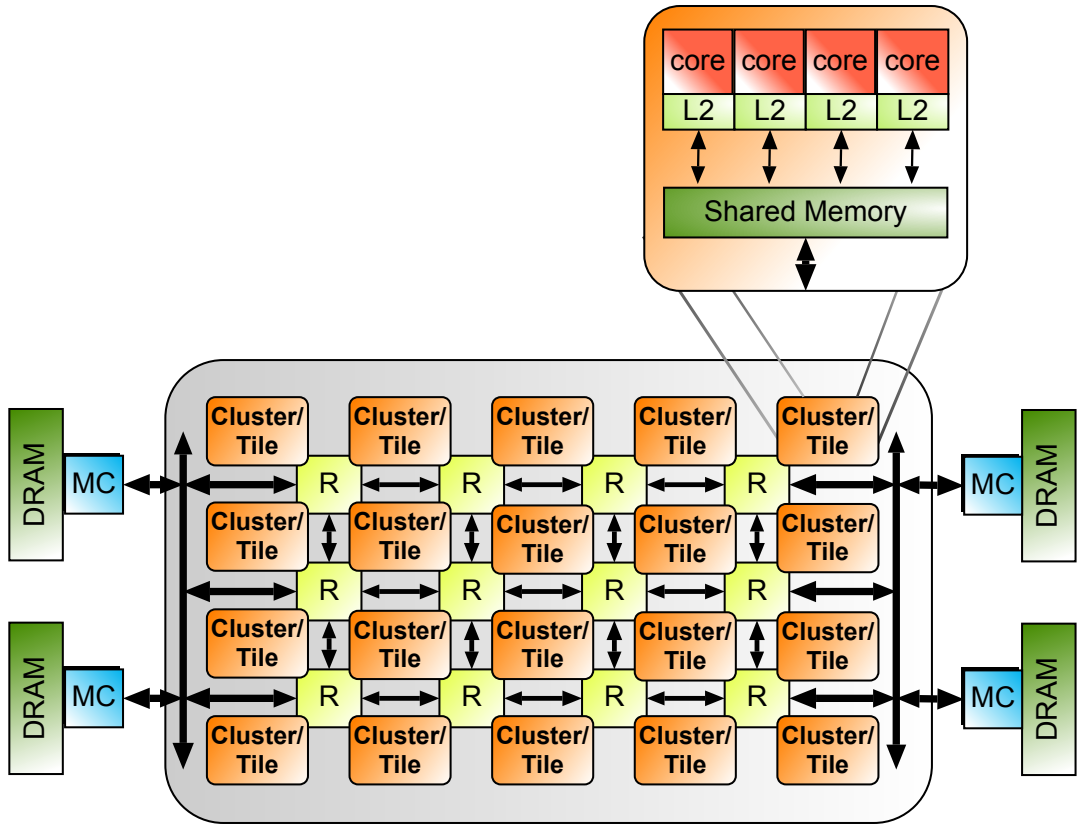


FIGURE 2.1: Abstracted many-core architecture

implemented as a *scratchpad*. One or more memory controllers on the NoC allow access to a large off-chip memory. Memory resources classified into two groups: on-chip and off-chip. Dedicated SRAM as on-chip in each tile and off-chip DDR-SDRAM modules accessible through memory controllers. The local memories associated with tiles are on-chip SRAM built-in and off-chip memory from the many-core memory hierarchy.

Tiled many-cores originated from the RAW research processor by MIT [19], later used commercially by Tilera [13]. Currently, several products exist that leverage the cluster-based design paradigm Table 2.1 summarizes a few representative instances, highlighting the main parameters.

The hierarchical interconnection system, and in particular the on-chip network at the top level, make memory operation directed out of one cluster subject to *NUMA* effects. Namely, the cost to access a specific memory location depends on the physical path that corresponding transactions traverse.

Such many-cores allow tremendous performance/watt improvements, at the cost of increased complexity in software and hardware [1, 28]. Therefore, effective programming abstractions are key to tackling the increased system complexity, aiming at delivering both ease of application development and effective usage of the system's huge available

TABLE 2.1: Real-world cluster-based many-core instances

Platform	Cores (Total)	Clusters	Cores/Cluster	HW threads
SCC [4]	48	24	2	1
MIC [12]	240	$\geq$ 60	1	4
STHORM [24]	69	4	17	1
KALRAY [23]	256	16	16	1
Parallella [25]	16	16	1	1
Tilera [13]	72	72	1	1

parallelism. In the next section, a brief survey of the programming models proposed for many-core systems.

2.2 Today's Programming models

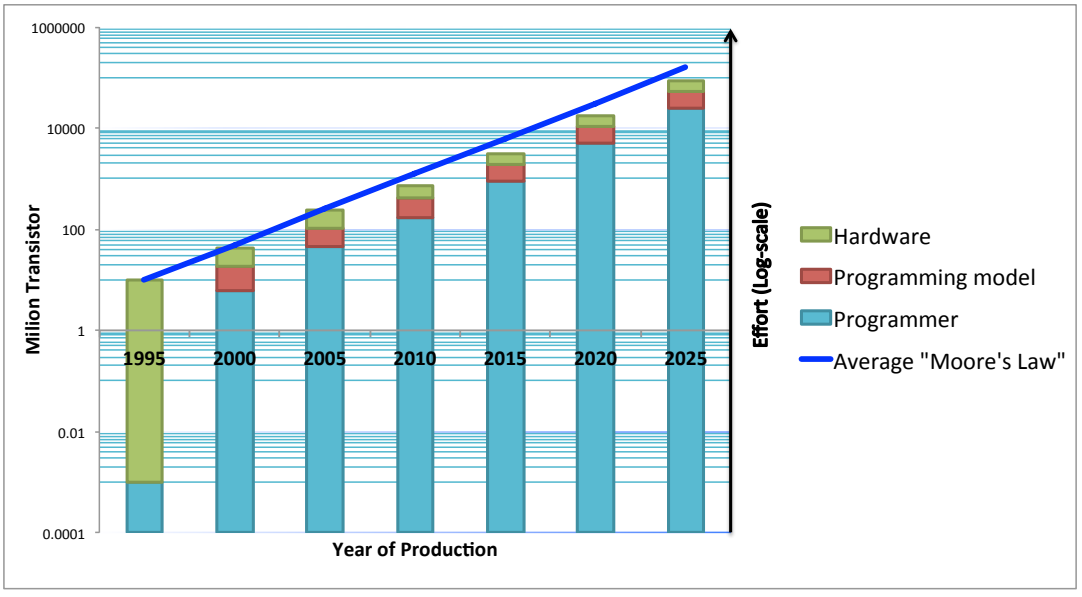


FIGURE 2.2: Programming Models of major many-core Platforms

Figure 2.2 shows the roadmap timeline for the effort of parallel execution design [2, 29]. The middle line depicts the average Moore's Law that now becomes:"*doubling the number of processing cores per chip every 2 years.*" Blue, orange, and green boxes in the figure highlight the effort of the programmer, programming model, and hardware, respectively, from 1995 as a reference year and through 2025. Prior to the late-2000s, the hardware implementation completely reduced the effort of the parallel execution activities from both the programmer (human) and the compiler as illustrated in Figure 2.2. Here, the hardware is responsible for executing instructions simultaneously and out of

their program order is hidden by a sequential retirement mechanism. Thus, programmers are a far cry from dealing with any complexity due to parallel execution. This model is dominated the parallel execution in general-purpose microprocessors and also the programming model for the majority of programmers today.

Since 2000, the parallel execution has been exposed at the machine programming level that added more effort to the compiler and the programmer. Where, we replaced the ILP model for programmers with a new parallel programming model that is known as *thread level parallelism* (*TLP*). TLP is developed by vendor engineers as the domain application programming interface (*API*), to obtain sustainable performance improvement. In this model, programmers divide up their applications into semi-independent parts (*threads*) that can operate simultaneously among the processors in a system. Note that the efforts of parallel processing are exposed to programmers when ever they need to take advantage of the processing power inherent in the multiprocessor design, since the hardware does not maintain sequential state. Namely, programmers need to understand the parallel execution model and to develop parallel algorithms, which be equipped with much better tools to automatically tune the performance of parallel applications. Of course, this requires education as well as incorporation with compilers to exploit identify parallel tasks. Regardless of these, still converting a sequential program is more challenging, as there can be developed to be easily in parallel. After 2008, the new approach is to implement a many-core system [30]. Therefore, programmers heavily rely on software tools such as *programming models*.

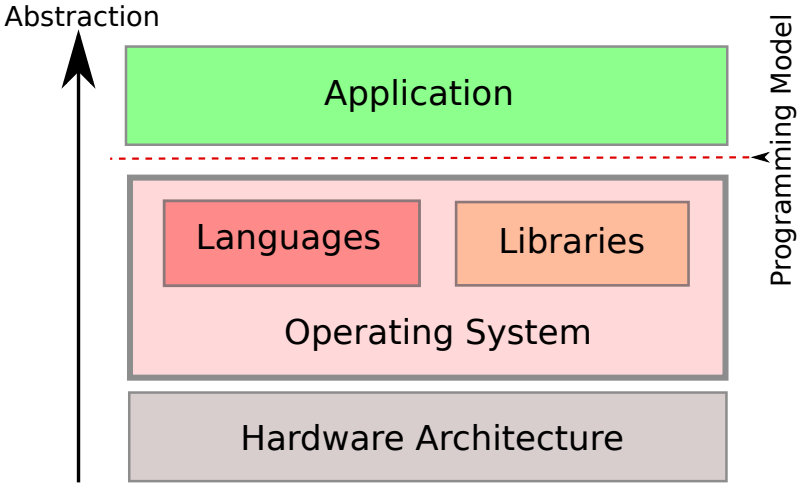


FIGURE 2.3: Programming model layers

As illustrated in Figure 2.3, programming models are an abstraction layer between the underlying hardware architecture and the software available to applications. It focused on increasing the developer productivity and achieving high performance and portability

to other system designs. In short, it allows programmers to focus on problem solving and correctness. Therefore, the programming model continues to be important because of the application needs to scale automatically with more number processors. One of the most interest approach in parallelism development is called *concurrency revolution* [31]. In this domain, parallel programming models can be distinguished based on the way that used to access to data: *shared memory* and *distributed memory* [32, 33]. In shared memory-based programming model that relies on the shared memory multiprocessors (i.e. SMP), allows to communicate by sharing the data in the global address space, to which programmers are accustomed. Therefore, shared memory-based programming model has proven to be very effective at simplifying application development. OpenMP [34] has emerged as a *de-facto* standard for shared memory system, since it provides a very simple means to express parallelism in a standard C (or C++, or Fortran) application, based on specific constructs. Namely, the programmer provides information on where and how to parallelize a program based on code annotations (compiler directives).

In distributed memory-based programming, it shows the memory address space is distributed for every processor that is a popular architectural model. Message passing model (*MPI*) is used very widely for distributed memory [35]. This model allows processors to use *message* as communication routines to exchange data among processors, where each processor typically has own private memory [36]. NVIDIA comes with new programming models (such as CUDA) to write massively threaded parallel programs for GPUs that supports data parallelism level [37].

Obviously, the many-core approach will be continue to be the choice of both industry experts [18] and academia [38] researchers by integrating hundreds or thousand cores on a single chip since it is the only way to scale performance from now on. For the foreseeable future, all of these programming models could serve as good machine-level programming, but unlikely they have cost-effective for the vast majority of application programmers. Because of writing software that can fully benefit from the new hardware architecture, that featuring 100 cores and more, much *harder*.

2.3 Anatomy of an OpenMP mapped on MPSoC

Recently, MPSoC consists of several computational subsystems (multiprocessors, or multi-cluster) are connected via a scalable communication medium (NoC) [24, 39] in a mesh, also is called a *tile* based architecture. Tile-based platforms have two kinds of fundamental interprocess communication models: shared memory and message passing. However, MPSoC represent a revolution in computer architecture that promising a

solution for forthcoming complex systems [40]. Nevertheless, due to the complexity increased of the MPSoC system with the presence of complex on-chip memory hierarchies and applications nowadays, has significantly complicated the production of software in such system development. MPSoC have more complexity than multi-core technology and provide higher performance, lower power consumption. As a consequence, it is no longer possible to discount challenges caused by converging the software and the hardware development [1].

Parallelism within a many-core system is most commonly exploited by using either a higher-level thread programming (MPI) approach or a slow-level thread programming like OpenMP. The purpose of these programming models is to extend the source language (normally C) to include multi-core features in a scalable way. This appealing ease of use has recently led to the flourishing of a number of OpenMP implementations for MPSoCs [41–45]. Many researchers have tried their best to employ or extend OpenMP into different MPSoC architectures (e.g, SMP NUMA system [46, 47], clusters[48], and even accelerators [49, 50]) so that the productivity of programmers is improved.

In the rest of this section, the techniques of implementation of OpenMP on MPSoC introduced and how the translation tool worked.

2.3.1 Translate the OpenMP Code

Once of the techniques used to execute OpenMP applications in MPSoC environment is to analyze the accesses to the shared data and generate a mechanism to handle the access to the data. At compile time, the OpenMP program can be translated and located the shared data access to other languages suited for target platforms, MPI or global arrays (*GA*) for example.

Basumallik [51, 52] and Millot [53] transformed OpenMP to MPI library. Wang [54] and Dorte [48, 55] implemented *LLCoMP* to translate extended OpenMP to MPI by using the skeleton method. This kind of transformations are feasible for the application with regular accesses, but the transformation becomes trickier in the irregular access satiation. Huang [56, 57] and Chapman [58] (based on OpenUH compiler) made the same transformation, but instead of using MPI, they used global arrays.

It will introduce briefly details about the traditional translation technique of OpenMP on GCC compiler in the next section, because it is essentially what any OpenMP compiler does.

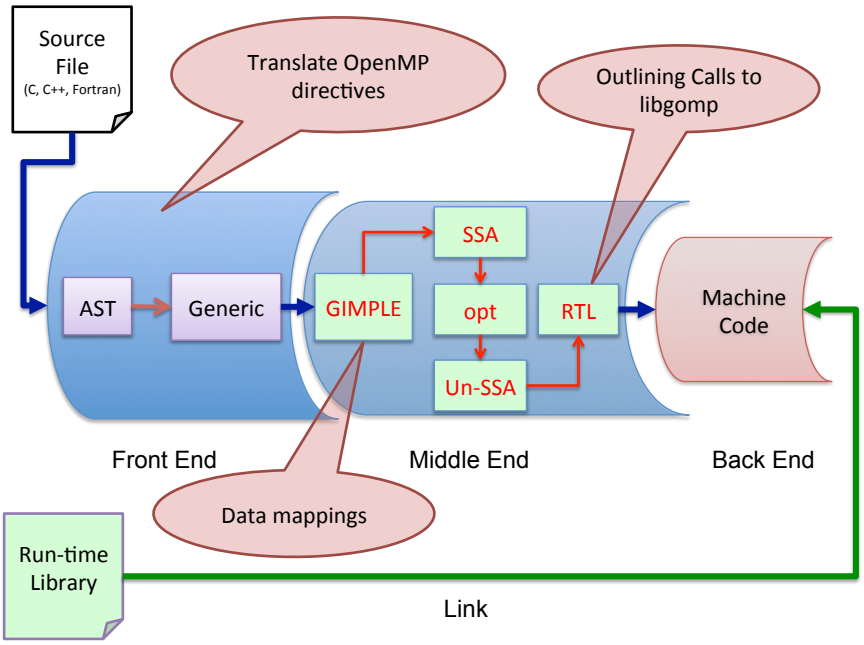


FIGURE 2.4: An overview of the GNU compiler

2.3.1.1 OpenMP in the GCC

The GNU Compiler Collection *GCC* provides OpenMP support for C, C++ and Fortran. The *-fopenmp* compiler flag using to recognize OpenMP pragmas. This flag will dynamically link the program with the GNU OpenMP library (*libGOMP*) [17]. Figure 2.4 shows the overview of the GCC passes. At compile time, all OpenMP applications related GCC code as highlighted in green, resides in the front end and middle end. Namely, the main OpenMP specific task at the front end, which used to parse OpenMP directives and clauses, check the integrity and generalize the directives to the GENERIC intermediate representation (*IR*) in the middle end[59, 60]. The files *c-parser.c:c_parser_omp_** *cp/-parser.c:cp_parser_omp_** and fortran/parse.c:parse_omp_* are used for the parsing and propagation of the directives for the front ends.

In a next step, the output of GENERIC transformed into GIMPLE IR. This transformation has done in *gimplify.c:gimplify_omp_** and *gimplify.c: omp_**. All implicit data sharing clauses and atomic directives are transformed into the corresponding functions. The gimplification using a special data structure that contains space for all non-global variables to passing them to parallel region. The *pass_lower_omp* in *omp-low.c* is used to create and fill the *data-sharing data structure*. It also inserts OMP_RETURN and OMP_CONTINUE instructions which are used to denote the end of a parallel or work-sharing region. Consequently, the IR is responsible for creating the *control flow graph* (*CFG*).

Therefore, GOMP could be used for MPI transformation for example, such as translating the OpenMP directives into MPI primitives as detailed in [61].

The following listings illustrate the lowering processes performed by GCC and the data structures used. In the Listing 2.1 as below, it shows an example of the most important OpenMP directive (*#pragma omp parallel*).

```
...
int a;
#pragma omp parallel shared(a) {
...
foo(a);
...
}
...
```

Listing 2.1: Example of a simple parallel region.

The output of compiler transformation shown in Listing 2.2, the compiler generates a data-sharing structure (*omp_data*) that containing pointers to shared data. Particularly, the IR stories shared variables address into a data structure and then passes the structure's address to start the parallel region (*gomp_parallel_start*). Then executes the parallel region and calls the function (*gomp_parallel_end*) to end the parallel region. The *num_threads* indicates the number of threads participation in the parallel region.

```
...
int a;
omp_data.a = a;
gomp_parallel_start(omp_fn, &omp_data, num_threads );
omp_fn(&omp_data);
gomp_parallel_end();
...
```

Listing 2.2: The GIMPLE representation.

Finally, the compiler replaces all references of shared variables within the outlined parallel function (*omp_fn*) with references to the corresponding fields of the data structure as illustrated in Listing 2.3.

```
omp_fn(omp_data)
{
...
foo(omp_data->a);
...
}
```

Listing 2.3: The outlined function.

As explained in Listing 2.2 and 2.3, the GCC compiler translated the OpenMP directives into outlined function (libGOMP runtime). Then, libGOMP is in charge of implementing the typical OpenMP API routines and low-level operations for the high-level constructs. Traditionally, this library itself is implemented by using POSIX threads. The address of omp_data structure is passed as an argument to the newly created function and all shared variable references within the function body are exchanged by the corresponding field reference (omp_data.reference).

2.3.1.2 OpenMP Runtime Library

The compiler links to an external library is called libGOMP. In particular, the libGOMP is responsible for thread management and the distribution of tasks of work-sharing among threads. LibGOMP exploits the POSIX threads (Pthreads) library [62] to create threads, which is available across many architectures and supports API for creation and manipulation of threads. There is a several struts such as *gomp_thread, gomp_team_state, gomp_team* and *gomp_work_share* which are used to manage threads. Furthermore, the main task of libGOMP is mapping OpenMP run-time library routines that dynamically create multiple instances of the outlined functions.

2.3.1.3 Transformation Tool

TABLE 2.2: OpenMP Compilers List

OpenMP Compiler	Availability
HP [63]	commercial
Fujitsu [64]	commercial
Sun Studio 12 [65]	commercial, free
MIPSpro [66]	free
MaGOMP[67]	free
OdinMP [68]	free
OMPi [69]	free
OpenUH [70]	free
Nanos Mercurium [71]	free
Microsoft Visual C++ [72]	commercial
PathScale[73]	commercial

There is another way to landing OpenMP on MPSoC environment by using *Source-to-Source* tool which takes as input C/C++/Fortran source code with OpenMP directives

and outputs equivalent to multi-threaded C/C++/Fortran code, ready to be built and executed on a system [68, 74–76]. The run-time library designed based mostly on the *outlining* technique [77] that is moved the code inside a parallel region into a separate function. Maybe the generated code is compiled by native back-end compiler (i.e. GCC) linked with the runtime library [78]. As illustrated in Section 2.3.1.1, the main OpenMP specific task is in the front end, which is responsible of parse and propagate the directives to the middle end. Therefore, It could use the source-to-source translation the front-end to generate an OpenMP programming model in the target architecture. For example, transforming the OpenMP into MPI as detailed in STEP project [53]. Also, there have been efforts to port OpenMP to the Cell B.E. [21]. The most successful one is the implementation in IBM's XL compiler [44]. Authors [57] implemented OpenMP on cluster by translating OpenMP programs to GA programs. This technique uses GA to handle the shared data and communication across different units in a cluster. In addition to GA, MPI library calls (MPI_Send and MPI_Recv) were used in the translation to guarantee the execution order of processes, which increased the complexity of the translated code. Finally, Table 2.2 summarizes a list of other experimental compilation systems, which do support of OpenMP implementation.

2.3.2 Translating OpenMP into Software DSM

Another technique is to use a DSM architecture that offers the abstraction of a shared memory layer between the different nodes creating a virtual unified address space. DSM run-time supports a model that unifies the message passing and shared memory programming models [79]. Since DSM's system spans both logically shared and physically distributed memory systems and allows parallel programs to use explicit message passing to translate access remote memory between independent processors, it has advantages. Firstly, understanding shared memory programs is easier and shorter than message passing programs, because the memory accessing is more popular. Secondly, remove the user's effort from any explicit awareness of communication. As well, it supports large virtual memory space [80].

However, DSM system has high latency when accessing remote data due to the overhead of the message-passing interface and the network access [81, 82]. To address this issue, DSM systems use a double buffer technique (to cache data from remote memory to local) to reducing the processor's memory access time and implement a *coherence protocol* that ensures a read by any processor to return the most updated data. A *memory consistency model* specifies how memory behaviours depending on prior memory reads and writes from multiple processors. Of course, the coherence protocol is dependent on the memory consistency model. However, several consistency models have been

proposed, such as *sequential consistency (SC)* was first defined by Lamport [83] that requires the memory operations appear to the execution as same as in the sequential order and the operations of each individual processor appear in the sequence of the order that specified by its program. While *Release consistency (RC)* [84] allows a programmer to leverage synchronization operations to create a partial ordering of memory operations.

However, several compilers working on transforming an OpenMP source code to its equivalent code to be used in a specific DSM automatically. As such implementation, the Omni compiler [76, 85, 86] that detects the shared variables and inserts the code to executed on top of the SCASH [87] DSM. Similarly, Intel has integrated TreadMarks [88] inside its compiler [89], which has promising results for small applications [90]. Also, some of research works [91, 92] translate OpenMP to MPI and DSM software to reduce the overhead of DSM system.

2.3.3 Hybrid programming

Using both of components at compile time and at run-time as option to support OpenMP. For example, ParADE [93], Min [94], and [95] use this kind of hybrid programming.

2.3.4 Partitioned Global Address Space (PGAS)

PGAS models support a global shared memory address space that may physically distribute to all participating processes. Examples of such models are Unified Parallel C (*UPC*) [96], Titanium [97], Chapel [98], and X10 [99].

In general, two different approaches are used to implement PGAS models, which expose locality to users in different address spaces: a *global address space* and a *local address space*. In the first approach, the global address space is partitioned in the logical manner into regions which are accessible to any process, regardless of where it is mapped (e.g. UPC). While in the second approach, only local partitions are accessible. However, in PGAS models, the accessing to remote partition has high overhead because the remote data must be fetched into local copy and written back into its location in the global address space. Therefore, the programmers are encouraged to reduce the remote partition access. Consequently, this model yields higher performance by offering direct control over communication, but sometime at the cost of more work on the part of the programmer.

2.3.5 Single System Image Hardware Virtualization

ScaleMP developed the Versatile Symmetric Multiprocessors (*vSMP*), a software based computing architecture, which combine a number of physical x86 computers to create virtually a single-system-image [100]. This technique is not new, it was already employed in parallel virtual machine (*PVM*) in 1989 to execute a large parallel application on a distributed computing system [101]. This hyper-visor creates a single operating system on multiple physical computers connected via interconnects and provides a unified virtual system to both the OS and the applications. While other hardware virtualization hypervisors such as Xen and VMware ESX, that allow multiple virtual hosts to work on the same physical computer [102, 103].

In fact, ScaleMP offers an alternative approach to run shared memory applications on distributed memory architecture. ScaleMP is similar to DSM systems in terms of handling communication by removing the explicit control of data exchange between compute nodes from the programmers. Moreover, ScaleMP handles the cache coherency between the individual units by using multiple advance coherency algorithms which operate concurrently based on real-time memory activity access patterns.

2.4 Related Work

In the recent past many researchers proposed implementations of OpenMP translator and run-time as suitable to MPSoC [41–44, 104]. One of the more interesting implementation is the Cell BE [44]. The Cell processor is considered as a distributed memory machine, where SPEs can only communicate with each other by means of DMA transfers from/towards the main memory [21].

Other researchers have implemented successfully OpenMP for embedded MPSoCs with a similar memory model. As such example as, authors of [41] present an OpenMP implementation for a Cradle CT3400 without an OS. An extended OpenMP programming framework for the Cradle 3SoC architecture is described in [42, 43]. They provided custom directives to exploit the memory hierarchy in the system. Authors [105] show the necessary extensions in the standard OpenMP to make it a valuable programming model for embedded MPSoC, by discussing an initial implementation for a TI C64x with a multi-level memory hierarchy. Authors [106] developed a mapping strategy that explores the opportunities to optimize OpenMP programs on the Cyclops-64. They showed that OpenMP as high-level parallel programming model for the Cyclops-64 platform by optimizing a memory aware runtime library, unique spin lock algorithm, and a

fast barrier synchronization. The authors of [107, 108] implement OpenMP on a dual M32R processor, which supports fully the POSIX execution model.

Marongiu [109] has been presented an extensive set of experiments and researches aimed at highlighting the challenges to support OpenMP programming constructs on a generic MPSoC template. Then, he proposed several implementation variants to reduce the cost of most common OpenMP programming patterns and also the extension of such implementation to a multi-cluster MPSoC. He also introduced techniques to support efficient data sharing among a very large number of cores (up to 64) [104]. Furthermore, he extended the directives and clauses to trigger array distribution across the memory hierarchy, which aimed to produce an efficient implementation of OpenMP by extending the API to support the exploitation of the memory hierarchy [110]. Additionally in [111], they support OpenMP implementation in both the host and the device sides by targeting the STHORM [24] architecture. In similar platforms, the authors in [112] presented an OpenMP task model that exploits a doubly linked queue to store the tasks. Using the task cut-of techniques and task descriptor recycling.

Lee et al.[113, 114] proposed an OpenMP-like programming models for easy MPI programming on distributed memory systems. They implemented OpenMPD that combined with explicit MPI coding to support data parallelism and work sharing paradigm which allow incremental parallelization for a sequential code. Then, they extended them effort to provide a new programming model has more flexibility to increase widespread of programming mode. XcalableMP (XMP) [114] is a directive based language extension of C and Fortran that includes data and task parallelism. Nomizu et al.[115] landed XMP on multi-node GPU clusters by adding news directive to handle data distribution between the host and GPU and OpenCL API to support various kinds of accelerators.

However, there is three challenges need to be faced when implement OpenMP on top of MPSoC, such as the MPSoC architectures, the complexity of memory hierarchy, and finally the synchronization implementations [109].

Chapter 3

The Single-chip Cloud Computer Architecture

Intel's SCC platform [4] is dedicated to exploring the future of many-core computing. It is a research architecture resembling a small cluster or "cloud" of computers, therefore, it is interesting in a variety of different applications through HPC and embedded domains.

3.1 Overall Architecture

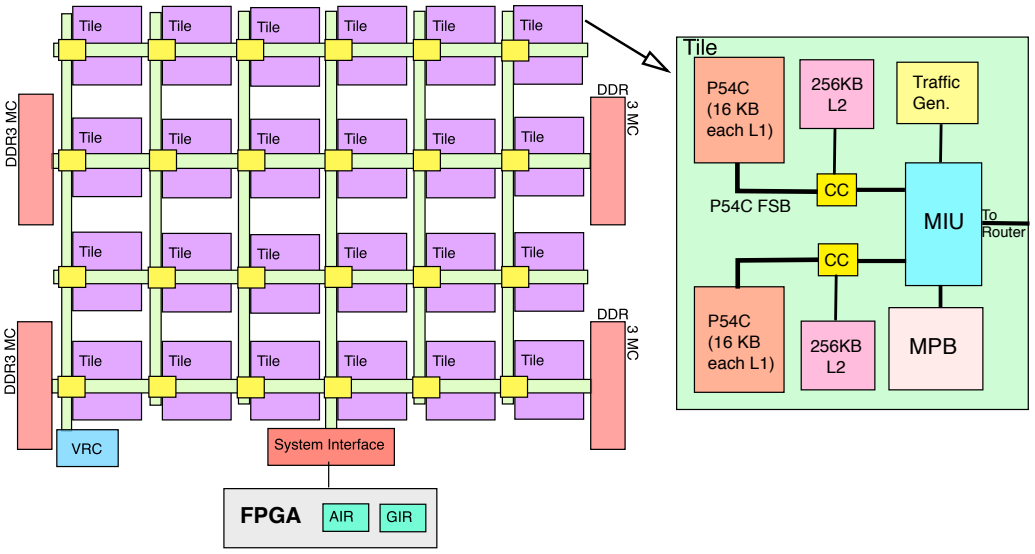


Figure 3.1: Layout and tile architecture for the SCC

As shown in Figure 3.1, the SCC architecture has 48 independent Pentium P54C cores, each one has L1 and L2 caches. Hence, P54C core can support compilers and

operating systems which require for full application development. These cores are organized as 24 dual-core tiles connected via a low-latency mesh network. The SCC chip is coordinated in a 6 x 4 grid and further decomposed into distinct voltage and frequency domains, some are configurable at start-up and others may be varied by applications during runtime. Each tile connects to a router and has two cores, a *Mesh Interface Unit* (MIU), and a pair of *test-and-set* registers for realizing atomic access. Moreover, SCC contains four on-chip DDR3 memory controllers (*MC*), which are connected to the 2D-mesh as well. Each on-chip MC supports up to 16GB DDR3 memory to provide 64GB in a total system capacity. In addition, the fast local memory located on each tile that is mainly utilized to facilitate message-passing communication between cores.

The SCC has a board management micro-controller (BMC) to control the entire system; it is responsible to initialize and shut down critical system functions. There is only one way to communicate with the SCC is through a 64-bit PC running the Linux operating system (MCPC) over a PCI-e interface. The MCPC has software provided by Intel Labs that enables developers to load operating systems and programs on any single core or all of the cores of SCC, manipulate the SCC configuration registers, to load from and store to the memory [116].

3.2 Memory System

Being based on the P54C architecture, it contains 16kB data and program caches with 32 byte line size and a 256kB private L2 cache, as shown in Figure 3.1. Each individual core is able to access only 4GB of memory that is divided into private and shared regions. Shared sections are potentially visible to all cores and by default the access is uncached because the SCC doesn't support any hardware cache coherence mechanism. To solve this limitation, each core has *Lookup Tables* (LUT) with 256 entries with 16 MB granularity translate the address mapping 32-bit physical core addresses for the 64GB system memory. It is part of the configuration register space that is itself mapped by a LUT entry and shareable between cores. Each entry in LUT is configurable and points to specific types of memory spaces (off/on-chip memory, configuration and synchronization registers). As a result, LUT supports programmers access to tile's configuration registers (such as *test-and-set* and frequency control), providing a rich fabric for software-managed policies.

MC is located at the edges of the chip (four tiles in the grid ((0,0), (0,5), (2,0) and (2,5))) as depicted in Figure 3.1. Each MC has at least two banks (DIMMs) with four ranks and is responsible for issuing data transfers by interleaving control sequences in-order for memory banks and ranks. As a consequence, the achievable bandwidth is

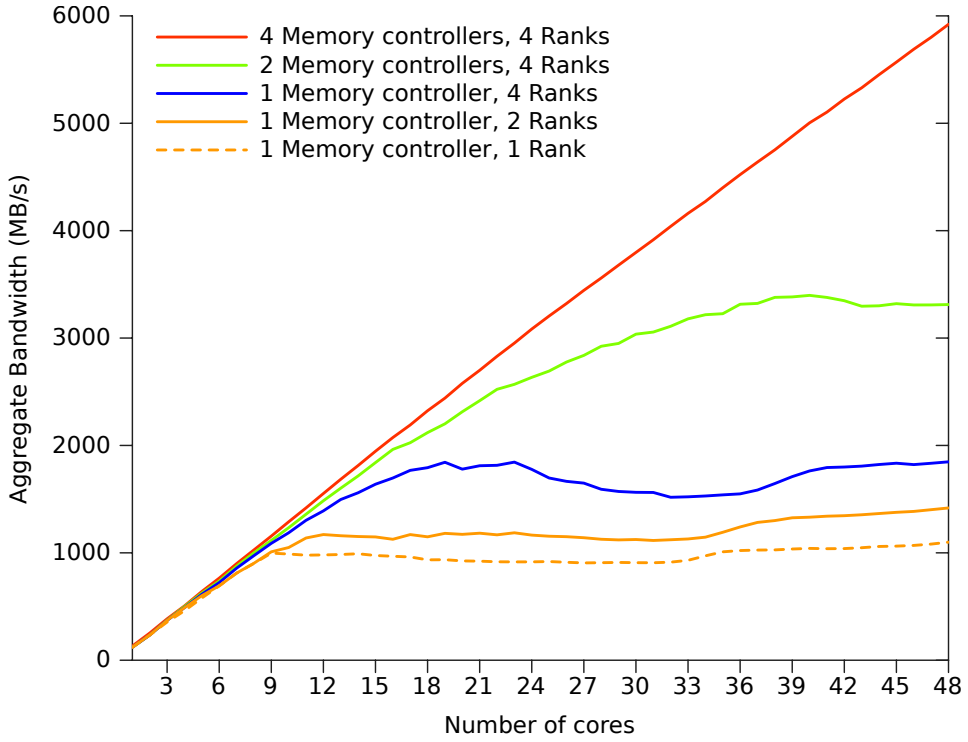


FIGURE 3.2: Total performance of four memory controller [3]

increasing because of the available number of memory banks per channel is high and interleaving of control sequences to different banks and ranks with closed-page mode. Figure 3.2 shows the memory performance with different number of cores per MC [3]. Obviously, an increasing number of ranks or MCs is contributing an increase in the bandwidth. In addition, there is another option to choose a configuration setting of memory subsystem to support different memory-bound scenarios by selecting different frequency settings, depending on the workload domain (communication-intensive, or computation-intensive) [117].

The system memory address space consists of 4 different 16GB regions of the external memory, 24 16KB regions of local memories (MPBs), and regions for memory mapped configuration registers of each core. The LUTs are usually set up at boot time and it can be changed dynamically when the system is running, having effect immediately. As a result, the data is shared between cores without needing to copy it. A core can map system-physical memory used by any other core at the granularity of these 16MB LUT pages [3].

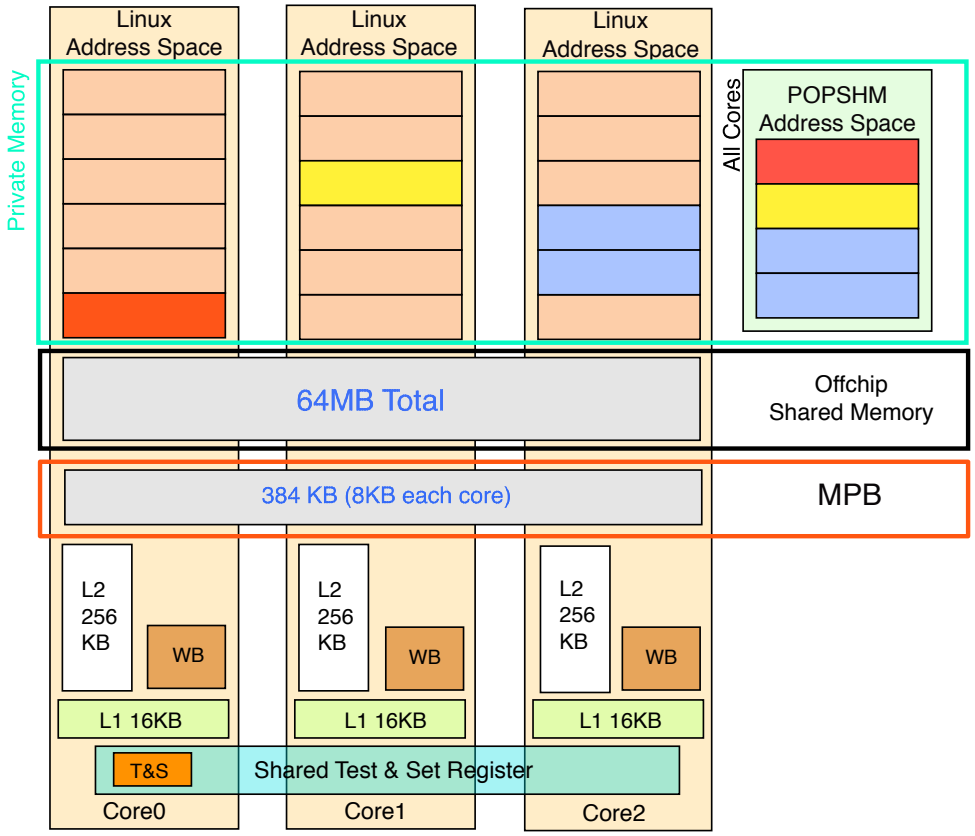


FIGURE 3.3: Shared address space in SCC

3.2.1 Hierarchy Memory

However, sharing the same memory region would cause a serious problem because there is no hardware support for cache coherence. Figure 3.3 illustrates the memory space defined by a core-individual lookup LUT, assigned to each of the 48 cores of the SCC, but accessible for read-write by all cores. The memory hierarchy consists of L1 (ICache, and DCache), L2, MPB, and shared main memory (off-chip). The LUT allows the programmer to map system-physical memory as shareable between the cores at the granularity of 16MB LUT pages. To be sure the core reading from the new target, the programmer needs to flush both the L1 and L2 cache, which is in the case of the L2 a very expensive operation. Because after remapping memory, a core might still read stale data from the L1 or L2 cache since these are indexed with core-physical addresses and reside before the LUT. Translation through the LUT entry needs a 40 core cycle ($\frac{1}{frequency}$) [116].

On the SCC, each core sees 64MB of shared memory (4*16MB chunks, each on separate MC), that can be accessed via non-cacheable pages via the *mmap* function call. Our run-time library statically decides on a location in the shared memory region to hold the shared data. To gain more out of shared memory, hijacking memory or

using *Privately Owned Public Shared Memory (POP-SHM)* that is used to allocate more physical shared memory. POP-SHM provides more space in shared memory by offering each core a part of its private memory and shares this with some or all other cores by exploiting LUT. Here, cores are grouped into four domains according to the four DRAM devices. The memory used in POP-SHM can only be accessed through the library such that the library can take care of flushing the caches when necessary. While in hijacking, the developer can increase the amount of available shared memory by using the unused LUT to move the boundary between shared and private memory [118]. Moreover, a Chinese Intel team have published Software Managed Coherency (*SMC*) [119]. SMC is open source and supports a virtual machine that provide a coherent, shared, virtual memory space for SCC cores. It handles the consistency on a page granularity based on the release consistency model [84]. After an acquire, a core issues a *smcAcquire()* to captures all writes from other cores and issues *smcRelease()* to publish it's updates at the point of release.

The SCC does not offer any cache coherency between the cores, but rather employs special 16kB-sized *Message Passing Buffer* (**MPB**) for improved communication efficiency between cores. The MPB is shared by all cores; in order to ease communication, it is partitioned into 8kB chunks for every core.

3.2.2 L1 Cache and Coherence Instructions

In Figure 3.4, a new **CL1INVMB** instruction together with a dedicated *message passing buffer type* (**MPBT**) are used to provide coherency guarantee between caches and MPBs. The flag (**PMB**) is used to enable the new memory type (MPBT). MPBT data is not cached in the L2 cache, but only in the L1 cache. Hence, when reading the MPBs, a core needs to clear the L1 cache. By using CL1INVMB, the core can clear its L1 cache lines that containing MPB data. As the SCC cores only support a single outstanding write request, a *Write Combine Buffer* (**WCB**) is used in MPBT mode to combine adjacent writes up to a whole cache line, it can then be written to the memory at once. A pitfall of the WCB is that *flush* data to memory is done only when another cache line is written to by a write instruction, or when the entire cache line is filled. Namely, the new cache line will become active in the WCB and the programmer needs a *dummy* write to a separate cache line to flush the WCB.

To update a data item in the MPB, one can invalidate the cached copy using the CL1INVMB instruction [120]. Resulting *write around* to the MPB and it never results in a hit in the L1 cache. By this hardware configuration, the SCC is designed to support the message-passing based programming models.

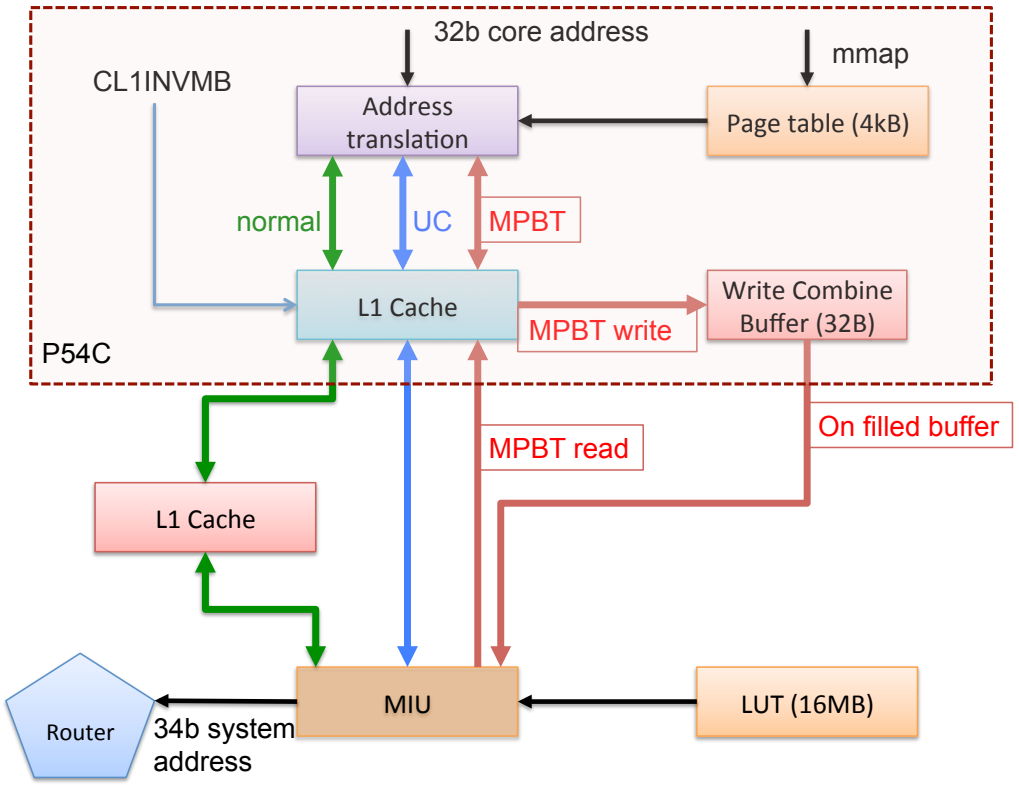


FIGURE 3.4: Address translation for P54C core on the SCC

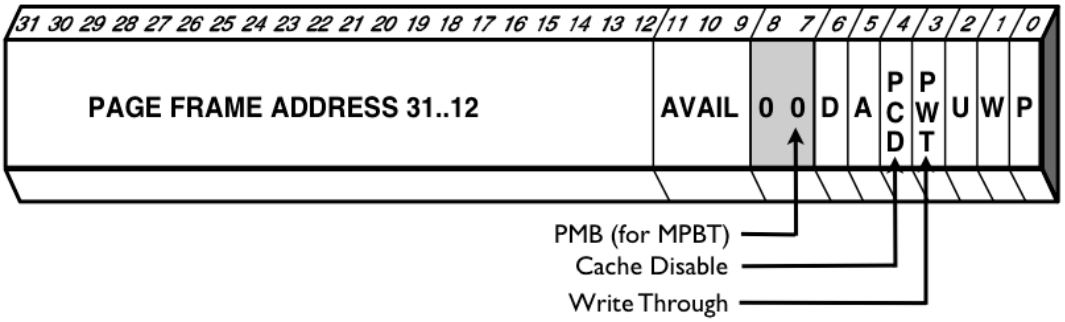


FIGURE 3.5: Page table entry for P54C architecture on the SCC [4]

Four memory modes supported by P54C SCC core that providing cache-related behaviour, *Un-cached*(**UC**), *MPBT*, *Write-Through*(**WT**), and *Write-Back*(**WB**). Figure 3.5 shows how these memory modes can be configured via bits associated with the page table entry. To change between those modes, Table 3.1 illustrates that the three bits which can change by setting or clearing them.

- Page Write-Through (*PWT*): Enable WT on the L1 cache. Namely, the data is put in the caches and in main memory as well in write operation and read the data directly from the cache.
- Page Cache Disable (*PCD*): Disable both L1 and L2 caches.

TABLE 3.1: Supported memory modes by setting or clearing bits

PCD	PWT	PMB	Memory Type
0	0	0	WB
0	1	0	WT
1	0	0	UC
1	1	0	UC
0 or 1	0 or 1	1	MPBT

- Message Passing Buffer Type (*PMB*): Enable the memory type (*MPBT*). Namely, the write goes directly to the WCB and do a single write to memory when the buffer is full. The L2 cache is always bypassed, as shown in Figure 3.4.

Moreover, the additional bit (11) in control register (*CR4*) must be set also to support the user access to MPB. While, the PMB bit can be set in the page table entry as described above and the developer must ensure that the Page Size Extension (*PSE*) bit in CR4 is disabled. Because it shares its location with the PSE bit that used for superpages and enabling both will generate a page fault. In addition, to be sure there is no any read or write L1 hits, one needs to invalidate and flush L1 first when changing data mode to UC.

The SCC core comes with the **WBINVD** instruction that can be used to perform a *write back* (flush) on the complete L1 cache, and **INVD** instruction to invalidate all lines in the L1 without written back. These instructions can only be issued from kernel-space. While CL1INVMB can be issued from user-space to invalidate all L1 cache lines which tagged with MPBT. To overcome this obstacle, the SMC library provides a nicer approach that can be used to change the access modes after the mmap() to any desired variant by extending the *mprotect* system call.

3.2.3 L2 Cache

The SCC designers added the 256KB 4-way write-back L2 unified cache to P54C cores that is placed external to the cores because the original P54C does not contain an L2 cache [4]. Namely, the write misses has to perform data writing directly to the memory which is fairly expensive. Because of the P54C cores only support one outstanding write per time and consequently the core needs to wait until the data is correctly written to main memory [3]. The cache line size is 32-byte, matching the cache line size for L1 internal to the core It needs a 10-cycle hit latency and several programmable sleep modes are available to support power reduction.

The P54C instructions invalidate/flush only the entire L1 cache or MPBT tagged data in L1 cache, and do not affect the L2 cache. The SCC has no native way to flush the L2 cache and the flush instruction therefore is ineffective [121]. The costs of flushing unmodified and modified data are reduced by an efficient way to flush the L2 cache to around 574K CPU cycles that implemented by the authors of [3, 121]. Unfortunately, flushing the L2 cache is still a very expensive operation. In addition, the user can set the cacheability for each individual virtual page space and turn off the L2 cache completely by set the PCD bit that disables both L1 and L2 caches.

However, the cache strategies are used in the SCC are *write back* and are *write around*, whereas the write miss did not allocate on. As a consequence, it is a very poor performance of *memcpy* operations because every write operation is issued as individual transaction instead of being combined into a full cache line write [3].

3.2.4 Memory Mapping

The SCC provides means to map pages by exploiting three special devices /dev/rckncm (UC mode), /dev/rckmpb (MPBT mode), and /dev/rckdcm (definitively cached mode (*DCM*)). Pages in DCM mode, the user can use the core's L1 and L2 caches by opening the device and using it as file in *mmap*() that used the physical address as the offset parameter. This mode access requires manual coherence management and special Linux object that provides a flush routine [122].

3.3 Tile Configuration Registers

Every SCC tile has a set of configuration registers. There is a pair of *test-and-set* locks registers that can be used to prevent race conditions by support atomic access to specific address.

In tile, each core contains a configuration register (*GLCFG0/1*). GLCFG0/1 has two bits which connected to the LINT0 and LINT1 pins of the local advanced programmable interrupt controller (APIC) for their corresponding cores. Theses bits are shareable between all the cores in the system and therefore can be used to trigger a hardware interrupt on any core. In addition, the other bits used to define the mode of core operation and status information from the external interface for the core [4].

To set the clock frequency of the local tile with its router, by change the 26 valid bits of the Global Clock Unit (*GCU*) configuration. The tile frequency is set between 100MHz and 800MHZ, while router is set at either 800MHz or 1.6GHz. This register

allows fine-grained power optimization with the Voltage Regulator Controller (*VRC*) across the chip. This register also has a bit that used to reset the core.

Each tile has a unique identifier in the form of (x,y) location in the 6x4 grid that contains in MYTILEID register. The value that got from this register differs by a single bit depending on which two core hold a seat on the tile reads it, identifying each core itself. As a result, the MYTILEID register can be used to differentiate code paths or parts for different cores via conditional jumps.

LUT0/1 is a configuration register that contains 256 entries with 16MB granularity to make up a core's 4GB physical address space. These entries are able to point to any system address including the LUT itself, enabling dynamic system memory mapping. Each core has access to any the LUT register in the system as well.

In addition, there is sensor registers (SENSOR and SENSORCTL) that allow to monitor and control the thermal sensors in the tile (cores and router). Moreover, L2 cache configuration register (L2CFG0/1) that controls the power behavior of the L2.

3.4 FPGA Configuration Registers

The bridge between the MCPC and the SCC silicon is the Rocky Lake system *FPGA* via the chip's system interface (SIF). It used to set up the chip environment, control applications, and develop MCPC applications. It allows users to set up the chip environment, control applications executing on the SCC and develop MCPC applications that communicate with the SCC chip. In addition, an external programmable off-chip component (FPGA) is provided to add new hardware features to the prototype. The off-chip FPGA in SCC offers additional registers which could used by cores to notify each other, which are: *a Global Time-stamp Counter* (**GTC**), *Atomic Increment Counters* (**AIC**), and *Global Interrupt Registers* (**GIR**) [123]. All those registers are accessed by memory mapping and the LUT. The GTC is a 64-bit counter that provides a common time base to all cores. It is available in form of two 32-bit values in registers and runs at the frequency of 125 MHz.

Every core in the SCC has a pair of AIC used as initialization and increment registers. Any read access to the increment register will trigger an atomic post-increment operation for the AIC value, whereas to decrement the current value by writing a value atomically. While a write access to the initialization register will initialize the AIC by 32 bit value and a read access simply returns its current value. The SCC's cores are able to send an interrupt to another core by writing a special value to the configuration registers of that core by using GIR.

3.5 Power Management

The SCC supports fine-grained Dynamic Voltage and Frequency Scaling (DVFS) infrastructure. In SCC, there is three components that works with different operating points which are selected as specified by their frequencies source: tiles, mesh, and memory DDR3. The frequency of the entire mesh can either be 800MHz of 1600MHz. When 1600MHz is chosen, the frequency of the MCs can operates either be 800MHz or 1066MHz. While, the frequency of the MCs is always set to 800MHz when 800MHz is chosen for the routers frequency.

However, mesh and memory frequency changes can be performed only during SCCKit program during booting time. The SCC cores are grouped in six voltage island with eight cores each, and voltage can only be changed in the scope of a whole voltage island. Similarly, so-called frequency islands can be set on a per-tile (2 cores) basis. Frequency and voltage scaling can be changed by using LUT mapping by each core for each voltage or frequency island. To adjust the voltage value that are stored in a special *VRC*, each core can send command to change the voltage on its island or another island. The VRC is a standalone part as depicted in Figure 3.1, therefore, the core needs at least three times to be sure the command has been finished [4], because the VRC handles one command per time only. In addition, VRC allows to adjust the voltage from 0 to 1.3V with granularity o 6.25 mV.

The frequency scaling is controlled by adding any integer value between 2 and 16 to configuration register that distributed among tiles. Consequently, frequency oscillate between 800 to 100 MHz and the maximum frequency is dependent on the voltage level. Any change in frequency register needs as little as 20 clock cycles to complete the actual changing. As a result, dynamic frequency scaling is faster and more flexible than voltage scaling due island size and because of he influence of frequency on energy consumption. It therefore can be applied in more variety of scenarios.

The power consumption of the chip is ranged between 25W (0.7V, 125MHz) and 125W (1.14V, 1GHz) based on the results from experiments performed by Intel.

3.6 SCC Programming Capabilities

Every 48 IA-cores of SCC does not feature a BIOS and boot own operating system (such as Linux) independently [124]. The Intel crew provide a modified version of the Linux 2.6.16 kernel that is capable of booting without a BIOS, it called sccLinux. Namely, all value obtained from BIOS are hard coded in the kernel, and other modifications

are based on timers and interrupts as well. In addition to using sccLinux, the user can implement an other OS on the cores to run customized application. Such as the Barrelfish operating system that is ported to the SCC [125], that matches the SCC hardware characteristics.

To control the memory, there are additional devices created in /dev, which are accessed by the rckmem driver. While, rckpc driver supports the virtual network interfacess for TCP/IP communication between cores or between the cores and the MCPC. Moreover, there are no Programmable Interrupt Timer(PIT), I/O APIC, and periphery like storage devices (keyboard or graphics card).

Therefore, the programmer needs a *MCPC* to communicate with the hardware. The MCPC holds the Intel-provided software (SccKit) that used to configure the SCC platform [123]. The SccKit contains command line tools and the *sccGui* and provides a tool that used for resetting cores, initializing the platform, accessing memory (DDR, MPB) and an API for handling I/O requests issued on the cores are available. Furthermore, the MCPC is used to compile the application(s) and port them on the SCC cores [126, 127]. Then, the MCPC using the shared directory to share the files with the SCC cores. This shared directory used as well to store or transfer the application executables and software extension packages respectively.

The Intel provides an MPI-like message passing interface, called *RCCE* [127, 128] that is used to explicitly managing MPBs for message passing. The RCCE is a small message passing library tuned to the needs of many-core chips such as SCC. The communication between cores occurs by transferring data from the private memory through the L1 cache of the sending core to the MPB and then to the L1 cache of the receiving core. As a consequences, the MPB allows L1 cache lines to move between cores without having to use the off-chip memory.

3.7 BareMetal

As mentioned before, the SCC allows the user to install and remove a specific operating system on any specific core. Furthermore, the SCC allows the programmer to use the chip without operating system that is referred to as *baremetal*. Two baremetal frameworks available from ETI and Microsoft. Microsoft developed baremetal environment package in which the user is able to run any code directly on the SCC [129].

E.T. International Inc. (*ETI*) provides a beta version of baremetal framework for the SCC [130]. It is a development toolchain for C programs with library support for libc, gdb, MPI, and MPB communication library. Programs are compiled into ELF

format and loaded onto the SCC via a utility running on the MCPC. One of the Intel community members made also a framework (BareMetalC) that can be use to run a simple C program directly on the system [116].

Finally, Micheal et al [131] proposed a minimalistic framework (BareMichael) that used for compiling, loading, and launching mixed C and assembly code on the SCC hardware. Furthermore, BareMichael is an open-source that has ability to load the same image of code to be loaded onto all cores at once, or redistributed different images to different cores, and delivering output through MikeTerm.

3.8 Related Work on the SCC

Many researchers developed and evaluated different programming models (shared memory and message passing) for the SCC platform in OS or runtime levels. All those models are improved with leverage to the SCC's hardware architecture. Supporting the shared memory programming model is non-trivial in such hardware because of the missing hardware cache coherency. Strictly, the situation will be tough when there is no virtual common address space between the cores in the system such as on the SCC. In this section, It is illustrated the most programming models which landed on the SCC platform.

The authors in [125] realize their implementation of the Barrelfish OS with a shared virtual address space over multi-kernel. This OS supports many parallel programs which are based on a model of many concurrent threads operating in a traditional shared memory space.

MetalSVM [132, 133] is a baremetal hypervisor that developed based on a shared virtual memory management system. It takes the responsibility of coherency management via the utilization of local memory on-chip.

The SCC features a disjointed memory space with hardware to support low-latency message communication. There are several message-passing parallel programming library which investigating how to best take advantage of that hardware. The Message Passing Interface (MPI) is a *de facto* standard for message-passing-based parallel model for communication in distributed memory systems [134, 135]. The RCKMPI [136] is a version of an MPI implementation that features three SCC-specific channels: *SCCMPB*, *SCCSHM*, and *SCCMULTI*. SCCMPB and SCCSHM use the MPB and off-chip memory for low level communication, respectively. While, SCCMULTI uses a combination of the two. Later, the authors in [137] took the benefit from user-supplied communication

technology information by reconfiguring the meta data in the MPB. Separately, the dynamic processes used to improve the SCCMPB channel[138], which a feature previously not supported in RCKMPI. Furthermore, Christgau et al [139] extended the RCKMPI by supporting the virtual process topologies. This approach improved the performance up to 44 % for a application that has intensively communication. The *SCC-MPICH* library [140] is another MPI implementation based on MP-MPICH [141].

RCCE [128] has a collective communication similar to the MPI standard. This library is more performance than RCKMPI and a less powerful API [139]. It consists of two/one-sided communication with primitives like RCCE_put and RCCE_get, and a power management tools such as frequency and voltage scaling [142]. The collective primitives such as *RCCE_bcast*, *RCCE_reduce*, and *RCCE_allreduce* are improved by many researchers [143–147]. Additionally, RCCE provides access to the shared memory space (64 MB) by remapping four LUT entries in each core to point for the shared space in each memory controller.

Clauss et al. [148] developed some useful extensions to the RCCE library that is called *iRCCE* (improved RCCE), which supports non-blocking send and receive primitives and a pipelined version of the blocking operations.

There is another message passing system (TACO) that is a distributed object framework [120]. TACO (Topologies and Collections) [149] is a C++ library that supported access to the SCC hardware and collective primitives that featured a highly efficient messaging back-end on the SCC. Then, MESH [150] ported on top of TACO to introduce direct access to shared data and consistency view for shared objects on the SCC as a middle-ware layer.

The SCC had already two other mechanisms to support shared memory. POP-SHM [151] provides two simple put/get primitives which used as interface to access shared memory. It extends the shared memory space by using a few LUT entries as read/write buffer in non-cacheable mode. The second is SMC [152] library that supports a coherent, allocation of shared pages, changing the access modes, and release consistency. Here, the programmer is responsible to choose data placement in private or shared memory.

The authors in [153] presented several techniques to provide a cache-coherent view of memory. All the techniques started with data reside in private space, and are only shared between all the cores if necessary. Here, the mechanism is similar for MESH framework [150], where the sharing is performed both through the shared of-chip memory as well as over the MPB, based on the nature of message being shared.

Kim et al. [154] proposed an efficient shared virtual memory as an alternative to the cache coherence mechanism for the SCC. They exploited the commit-reconcile and

fence (CRF) memory model to implement the shared virtual memory protocol. Here, the compiler or programmer is responsible to identify the data that should has consistent view between the cores. In addition, this implementation does not maintain twins or have any process for making diffs. It needs just to copying the data between a private memory space and the shared memory, according to a simpler protocol.

MapReduce is the most popular programming platform for data-intensive computing. The authors in [155] provided a scalable implementation of MapReduce that effectively utilize the NoC and local shared memory. In addition, this runtime highlighted the scalability bottlenecks of MapReduce and linear scaling of apllication with realistic datasets for a single SCC core. Although the promising performance results, the implementation of task scheduling and the design of full MapReduce with application analysis are still questionable.

The authors in [156] have ported S-Net to the SCC. This framework simplifies the parallel computation simply by describing data dependencies. The most interesting part in [156] is the comparison of the runtime of the different cache and memory policies for sending messages using shared memory on the SCC. Here, shared memory implemented by remapping the LUT pages, consequently making it possible to write messages directly into the cores memory.

In [121], the Self-adaptive Virtual Processor (SVP) model is implemented that is an abstract of concurrency programming model. The SVP can be used to express concurrency at many levels of granularity and uses shared memory semantics with weak consistency model. The authors in [121] ported the distributed implementation of the SVP [157] by using different communication approaches to more efficiently use the hardware messaging support on the SCC. Here, they employed several of the techniques to copy memory efficiently such as iRCCE, memory remapping, and dedicated copy cores. A bit similar to this work introduced by Prell et al. [158] which presents an implementation of Go's concurrency constructs on the SCC. Go-routines describe concurrently executing functions or computations in general. Here, the programmer is encouraged to "shared the memory by communicating" instead of to "communicate by sharing memory". Namely, the safe concurrency is achieved by using channels as a way to communicate and synchronize based on the message passing protocol. The channels implemented directly on the MPB, and the number of channels that can be concurrently utilized are limited because of the size of MPB is small.

Lee et al. [159] supported OpenCL [160] framework (runtime and compiler) on the SCC. The OpenCL's coherence and consistency model implemented on top of the SCC's message passing hardware, and modifying the control registers of each core to transfer memory blocks between the cores without using any expensive memory copy operations.

It is the first work for building a transport software layer to improve ease of programming and to achieve high performance.

TFluxSCC [161] is one of interesting project that exploit the parallelism in SCC by supporting TFlux Data-Driven Multithreading model. Here, the TFluxSCC system used a source-to-source compiler to translate the C program augmented with directives (threads and their dependencies) to appropriate runtime call to deal with the threads scheduling in a Data-Flow manner.

In addition, many of other projects are developed to support the distributed object on the SCC such as X10 [162] and MCAPI [163].

Best to our acknowledge, our OpenMP implementation is the first work for building a complete programming model to translate OpenMP code to leverage the hardware resources of the SCC. Our approach implemented from scratch and using the memory system as a flat shared memory to support the shared data among the cores by compiler extension. Furthermore, this approach designed in low overhead and high scalability runtime to give programmer a shortcut and an easy way to write his application.

3.9 SCC Setting

In the SCC system, the experimental results are generated using the default SCC settings, which are standard LUT entries, 533 MHz tile frequency, 800 MHz mesh and DRAM frequency for all micro-benchmarks. The experiments are conducted using SccKit 1.4.2.2 running a custom version of SccLinux based on the 2.6.32.24-generic kernel. For timing analysis, **RDTSC** (Read Time Stamp Counter) instructions [164] are inserted before and after the functions to be measured. The cores of SCC are single-threaded. In the rest of the thesis, it is considered cores and threads to be equivalent, as we do not "oversubscribed" cores but only assign one thread per core. However, all the experimental results in this thesis are generating based on the setting that explained in above.

Chapter 4

Tackling the design of the OpenMP Model

Systems-on-chip (SoCs) will constitute the best way to cover an increasing number of cores in a single chip and will continue for at least another decade based on Moore's Law expecting [6]. With systems featuring 100 cores and more being on the market, they typically provided higher performance with lower power consumption and more complexity than multi-core technology. Consequentiality, effective programming models tackle new challenges due to the effect of scaling [11]. Nowadays, the software stack is responsible for making effective use of the systems' resources to hold tremendous peak performance. Of course, this requires to employ all processors for most of the time and on-chip memory will also be distributed that have NUMA behavior. It therefore is necessary to provide coordinated execution efficiency of a multi-threaded application on the system cores.

Namely, programming a many-core system is difficult, specifically, if the system has a user-managed memory hierarchy, e.g. the SCC. OpenMP is a widely used parallel programming as solution for multi-core architecture. This programming model currently is used to decompose the computation code (e.g. loop iterations, tasks, etc.).

This chapter will go over the design of the OpenMP execution and memory model for SCC, describing my initial experience with the GCC compiler and a custom implementation of the run-time library. Specifically, It first studies the parallel code generation for OpenMP by GNU GCC. It then describes the design of the SCC OpenMP run-time library by coping with three challenges in the system:

- Supporting unmodified legacy OpenMP programs on SCC.
- Implementing the OpenMP memory model.

- Reducing the overhead for synchronization directives
- Analyzing the overhead of the fork/join implementation.

4.1 OpenMP Model

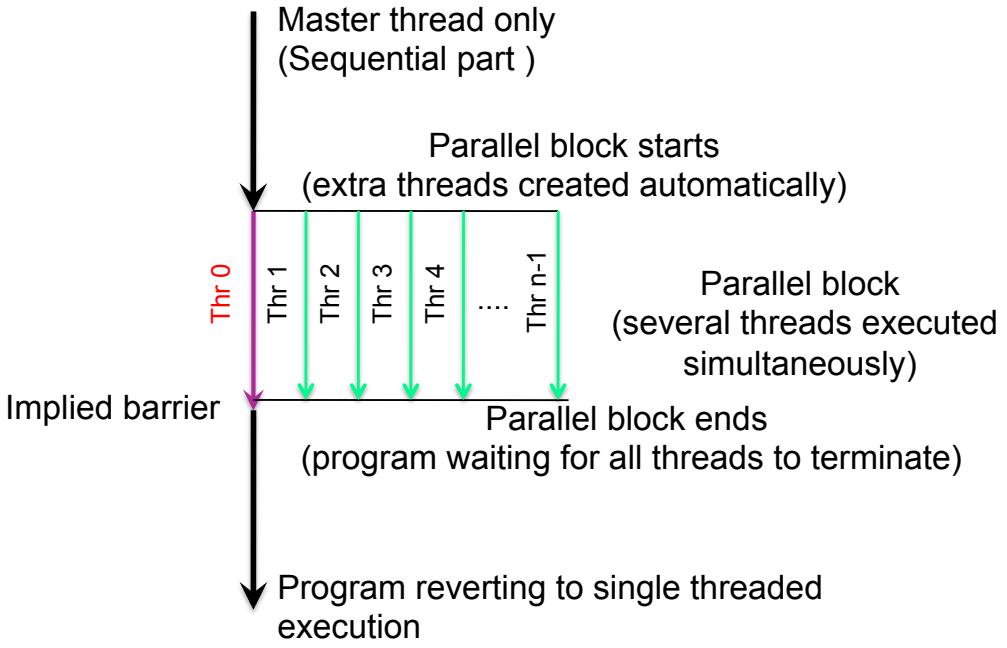


FIGURE 4.1: OpenMP fork/join parallel mechanism

OpenMP (and most related shared memory-based programming models) depends on a *fork/join* execution model, which uses a *barrier* construct to synchronize parallel threads as shown in Figure 4.1. Barriers – implicit or explicit – mean central constructs to any shared memory parallel program and to the OpenMP execution model.

OpenMP [16] employs the *fork/join* programming model as an easy and flexible way to handle sequential and parallel parts of an application. The program executes sequentially within a single thread, referred to as the *Master thread*, until it encounters a `#pragma omp parallel` directive. Here, program forks into a multitude of threads by assigning (forking) the computation into a number of worker threads (*slaves*). As a result, a parallel region is generated. At the end of the parallel construct, the master waits for all slaves to complete (join) before continuing execution. Then only the Master thread resumes execution again. A *barrier* is used in the end of parallel region to ensure all slave threads have completed before the master thread can continue.

A conventional OpenMP system consists of two major components, an OpenMP compiler, and a run-time library. In the remainder of this section, the challenges and needed modifications to the compiler and run-time design are discussed.

4.1.1 OpenMP Translation

In my approach, code transformation and calls to the run-time library are automatically instantiated by a customized GCC 4.6 compiler. GCC was used as a starting point as it provides open source implementation of the OpenMP translation pass and run-time library (**libGOMP**) [17]. The pass of libGOMP integrated in the complete GCC compilation and

Figure 4.2 explains the transformation process of the compiler using the sample code. Here, the compiler alters the execution flow of the original sequential program into a compiler-generated function (function outlining). Using the `-fopenmp` flag enables OpenMP translation, and dynamically links the transformed program to the libGOMP library. All OpenMP applications which related GCC code as highlighted in blue, reside in the front end and middle end, as depicted in the figure. Namely, the main OpenMP parallel construct (e.g `#pragma omp parallel`) in the front end that used to parse directives and clauses, check the integrity, and generalize the compiler annotation to the middle end in the GENERIC IR [59]. This generation strategy consists in outlining the parallel region body into separate functions used as an interface to libGOMP. The compiler adds an additional parameters into outlined function such as the loop iteration bounds for parallel loops, so that each thread only computes from the lower bound to the upper bound.

However, `#pragma omp parallel` blocks are outlined into new functions containing the code to be executed by parallel threads, as illustrated in Figure 4.2. The compiler encapsulates all shared data into a C-like `typedef struct` and inserts a call to the run-time function (`GOMP_parallel_start`), passing the new function and the struct pointer as arguments. This allows new threads to execute the parallel function and to point to the shared data items. To end the execution of a parallel region for the master thread, the compiler inserts a call to the `GOMP_parallel_end` function that contains a primitive of barrier synchronization to provide efficiently coordinated execution of the parallel threads.

In the original design of libGOMP, POSIX thread (Pthreads) is used as a standardized API for creation and manipulation of threads within certain operating systems (e.g. SMP GNU/Linux). Using *Pthreads* on many-core systems such as SCC would need dedicated abstraction layers to make possible the communication between threads on different

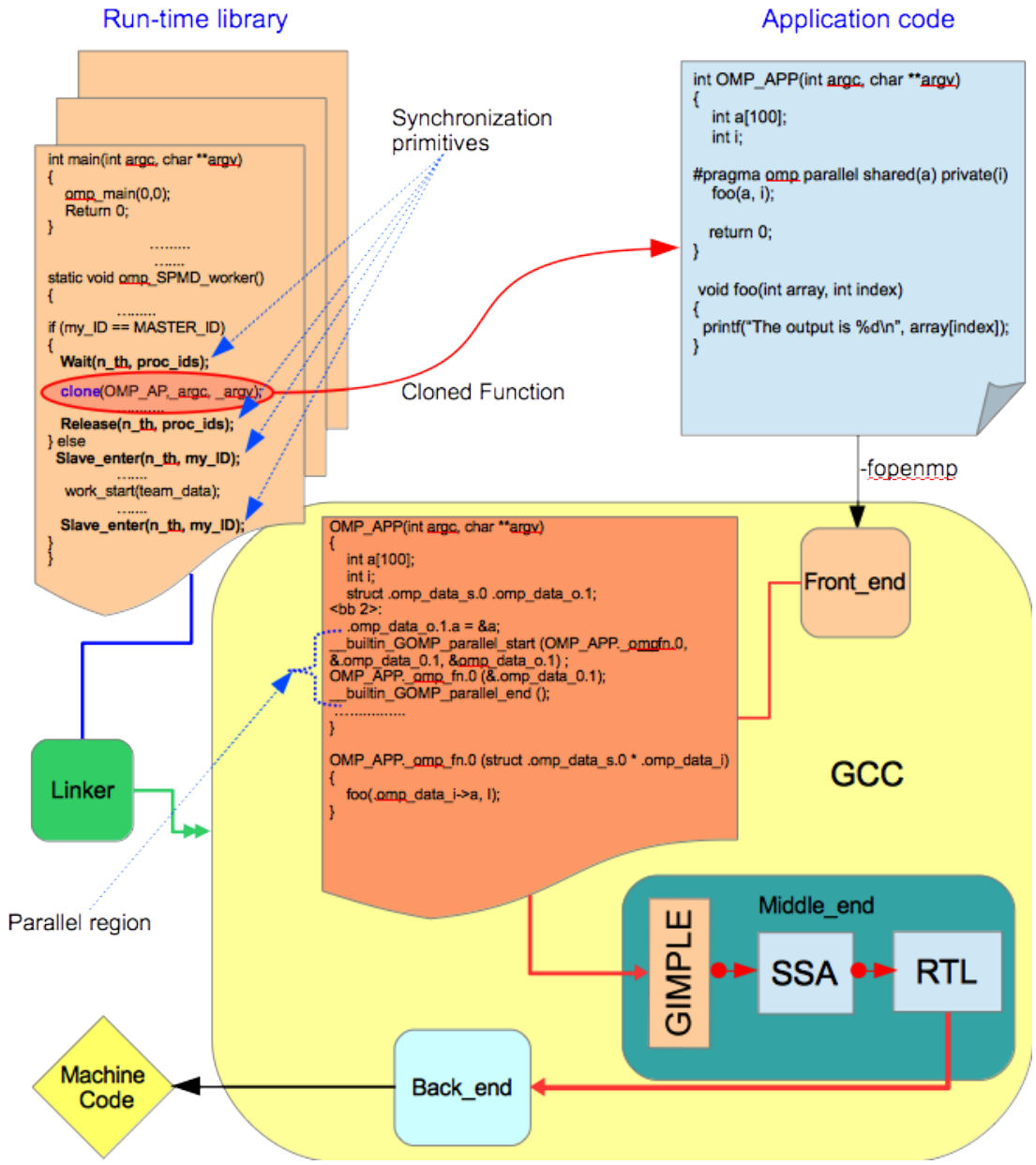


FIGURE 4.2: OpenMP code example and GCC compiler transformations

cores. Furthermore, the overheads that associated with the library (such as conditional variables and signal handling techniques) and as context switching that takes hundreds of cycles to execute [165].

To prevent these inefficiencies from limiting the parallelization effectiveness, the new run-time environment (**libgomp_scc**) designed from scratch in order to efficiently organize the parallelism of OpenMP applications on the SCC platform. *libgomp_scc* is a low-level library-based API that has the capability to manage resources in SCC system. In a traditional OpenMP implementation, the master thread is responsible for creating parallel worker threads when encountering a parallel block and for tearing them down

upon the end of the parallel block [166]. Dynamically creating and destroying threads that a parallel construct is encountered every time, it is very costly, so one opt for a different solution. In the implementation of the *libgomp_scc*, a custom micro-kernel code executed by every core at start-up [105, 167, 168] by assuming a fixed allocation of the master and slave threads to the processors. As a result, the threads can be quickly re-started at a later time when threads are docked upon parallel block end. Specifically, *persistent threads* created at program launch by loading the executable image of the program with library onto each processor (local L2 memory) at boot time.

Then, each thread (master or slave) runs the library code based on their core IDs. Since each core in the system had own OS, the developer needs a way to passing the pointer of outlining function and its arguments. Fortunately, The ELF standard has the header sufficient flexibility that used to define an arbitrary number of sections, to facilitate easier dynamic linking and debugging. One of the most efficient areas is Global Offset Table (GOT) that stores the absolute location of a function calls symbol, to allow the code to access the address of the variables which are not known at compile time. So, the dynamic linker (it is part of the operating system) resolves the GOT entries when the program starts. As a consequence, rather than copying or cloning the outlined functions in shared memory, the compiler just generates code at compile time and sending the pointer of the outlined function to the slave. As for the function arguments, it will discuss in more details in Section 4.1.3.

4.1.2 Parallelism Model

Parallelism model of OpenMP is based on the fork/join model as shown in Figure 4.1. As illustrated in Section 4.1.1, the GCC compiler translates any OpenMP directive into multi-threaded code containing function that calls a customized run-time library. Then, the run-time used to map the OpenMP parallelism onto the SCC architecture. To do so, interaction with the OS in the implementation of the runtime is hid, thus abstracting architectural concerns from the programmer's view. Currently, OpenMP mostly has been adopted the Micro-tasking Model [169] to implement the fork/join model. In this model, the master thread is only responsible to handle the creation and execution of parallel function. Traditionally, conditional variables and signal handling techniques are used to synchronize threads in the original implementation of OpenMP. The POSIX thread library presents conditional variables that require a thread be waiting for the conditional variable to receive it. Meanwhile, the core cycles are released and can be scheduled for another task. One of the main drawback of this schema is the larger context-switch overheads between the sleep and wake-up states as is the scheduling policy of GNU/Linux [170].

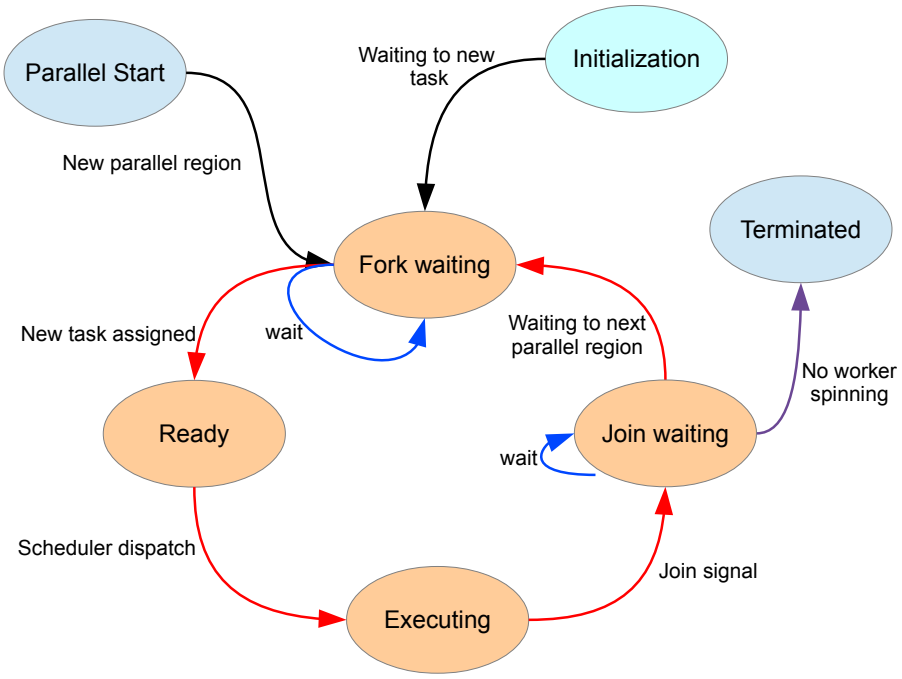


FIGURE 4.3: State transition diagram of OpenMP on the SCC

To work around this problem, a simpler *busy_waiting* mechanism is relied. Among many implementations, one that generated much interest is that of the authors at the University of Bologna [104]. The authors did not implement the OpenMP based Pthreads library, since the implementation has limitation to a single core, and the overheads associated with library (as overheads on context switch) and system bios, although using the Pthreads based run-time library required minimal modification. They used a fixed allocation for the master and slave threads to the processors, to minimize the cost associated with the dynamic thread creation, or multitasking libraries.

Figure 4.3 shows the state transition diagram for the OpenMP implementation on the SCC. As shown in the figure, there is a total of seven states. In the *initialization* state each processor loads its executable image, it contains the program and library, onto its own local L2 cache memory by taking advantage of the direct-mapped L1 program cache. Master and slave threads are executing different code based on the hosting core IDs. The master begins the execution on the main program, while the slaves wait to receive a notification from the Master thread about available parallel work. When the Master thread encounters a parallel region during program execution, the master is set to the *fork waiting* state, where it recruits slaves for parallel execution (as many as the user has specified),as explain in Section 4.1.4. After that, the state changes to *ready*. Here slaves are indicated the parallel function and the shared data. This activates the execution on the slaves, and the master itself transitions to the *executing* state. When the execution is complete, slaves enter the *join waiting* state, where they busy-wait

for a new parallel work or for the master to terminate the program. Here, the global barrier synchronization used implicitly at the end of the parallel region. Authors in [171] suggested to use a message exchange mechanism, where the slave cores spin on a local queue, to be sure there is no overlap between the execution of sequential sections of the program on the master thread.

Note that it uses distinct memory locations for different threads to busy wait (polling). Thus, each thread is waiting for its own task (new) rather than competing for global tasks. As a consequence, the system doesn't need to swap between sleep and wake-up modes every time, thus the overhead of context switching is reduced.To implement the spin lock, small storage cells into local message passing buffers is used, thus the busy-waiting cost and the contention for the system interconnect are minimized.

4.1.3 Memory Model

OpenMP comes with a relaxed memory consistency model similar to the weak ordering memory model. In this model, each thread have a *temporary view* of the memory that it should use to store data temporarily and hidden to other threads. Writes to memory are overlapped with other computation and reads from memory are satisfied directly with a local copy of memory, until it is forcing into memory by OpenMP *flush* operation. Obviously, such a memory model can efficiently implement on SCC memory system, since SCC has local memories which are usually accessible with different address space and cannot be accessed by other core directly [154].

In OpenMP, there are two main data qualifiers: *shared* or *private*. A shared variable is accessible by all the threads inside a parallel region while the private data has a distinct instances (one per thread) of a same variable. The OpenMP memory model suppose the shared memory space is a single and flat.Therefore, on SMP machines, multi-level coherent caches are used to reduce the cost of the memory access while preserving the abstraction of a unique memory space. The SCC (in most MPSoC) has no cache coherence between processors, alternatively it introduces an on-chip memory system to reduce the latency of memory access. It is currently working on efficiently supporting data sharing on SCC. The main difficulties of such design are summarized in the following two subsections.

4.1.3.1 Local Memory

Indeed, the first issue will arise when trying to run an OpenMP application on MP-SoC architectures, is OpenMP shared data. The OpenMP shared data may include

subroutine-local variables (*auto-variables*), which by default placed on stack (local private memory). Practically, the stack on one core is not accessible by others, because of every core-local memory has been mapped to different address space. Therefore, to use shared data in the stack, should be there is a way to make it visible to other cores. Listing 4.1 illustrates a shared data semantics in an OpenMP program.

```
int a;
void foo(){

  int b, c, d;

#pragma omp parallel shared(b,d) private(c) {
a = b + c + d;

}
```

LISTING 4.1: Illustration of variables visibility.

Global variables, like **a** in the example, are considered to access globally by each thread in a SMP system. It is by nature private to each core, as a consequences, it is used to handle *threadprivate* variables. For this reason, the original GCC implementation references the variable by name within any parallel thread is sufficient. Also, it doesn't even need to declare the variable as `shared` with the `parallel` construct. On the contrary, the whole address space on SCC is aliased over different processors by default except when it declared as shared variable. As a consequence, referencing a variable at a given address from two cores is causing in accessing different physical memory locations.

Non-global variables are declared within the scope of the sub-routine which contains the `parallel` directives, mapping on the master thread's stack. Even in a SMP system, threads can access each other's stack, it is necessary the code is generated to pass the variable by reference. While on SCC this is insufficient, since the stack of each processor is allocated to private local memories, which in turn are accessible through an aliased range of the global memory map (i.e., the same address on different cores addresses to different physical spaces). This is the case of variables **b** and **d** in the example.

Variables declared as `private` in OpenMP imply that the parallel thread owns a private replica of that object. The GCC compiler implements this by replicating the variable declaration within each parallel thread. In the SCC, it doesn't need to modify this behavior, since private data allocates by default onto local memories to each core.

One can change the GCC design to deal with global variables in a similar approach to what is done for automatic data declared as `shared`. But, it still has problem of sharing pointers between distinct threads.

There are several techniques to tackle this issue. One is to use CPU-specific handling technique of stack frames [68]. This scheme has two disadvantages. First, the generated code would no longer be similar for the two execution models. Second, it is clearly slow and complex.

OpenMP* run-time library supported technique that receives a separate pointer of every shared variable in outlined function [172]. This schema has performance disadvantages for parallel regions with many shared variables. Another option [168], is that to augment the original GCC mechanism to marshal shared variable within a structured construct to pass continuously shared objects by reference. This eventually allows to overcome the issues which related to memory aliasing when referencing data by name. However, for this approach to work, the program data allocated necessarily in a portion of the physical SCC shared memory that can be ultimately made unequivocally addressable by different threads. Currently, the main drawback is the non-coherent design of the transactions involving the shared memory.

4.1.3.2 Non-Coherent Cache

The SCC provided physical shared memory without cache coherence. Therefore, the latency-access to shared data should be a very high compared to accessing to private local memory. Fortunately, the memory model of OpenMP needs a coherent view for shared variable only at specific synchronization points because of relaxed consistency memory model. Thus, is possible to manage the cache coherence by implementing a specific flush instruction in the runtime [153]. Another approach can use a software-managed cache. Allocating data on the on-chip local memory (MPB) could also be explored, but the limitation in the size and the explicitly data transfers are making it a less appealing solution. In the other hand, reducing the shared memory access by putting some data into the on-chip memory (MPB).

4.1.3.3 The Solution

One promising solution is to use the main shared memory to achieve efficient data sharing on SCC. Possibly extending the programming interface with custom directives to place shared data items of a program in this memory region. Practically, this shared memory has little physical space, so it is also exploiting the possibility of using part of the cores' private memory (e.g. hijacking or POP-SHM). By leveraging LUT registers to resolve virtual addresses and to modify the compiler in a way to pass shared data offsets instead of pointers. Here, one basically needs to follow this procedure:

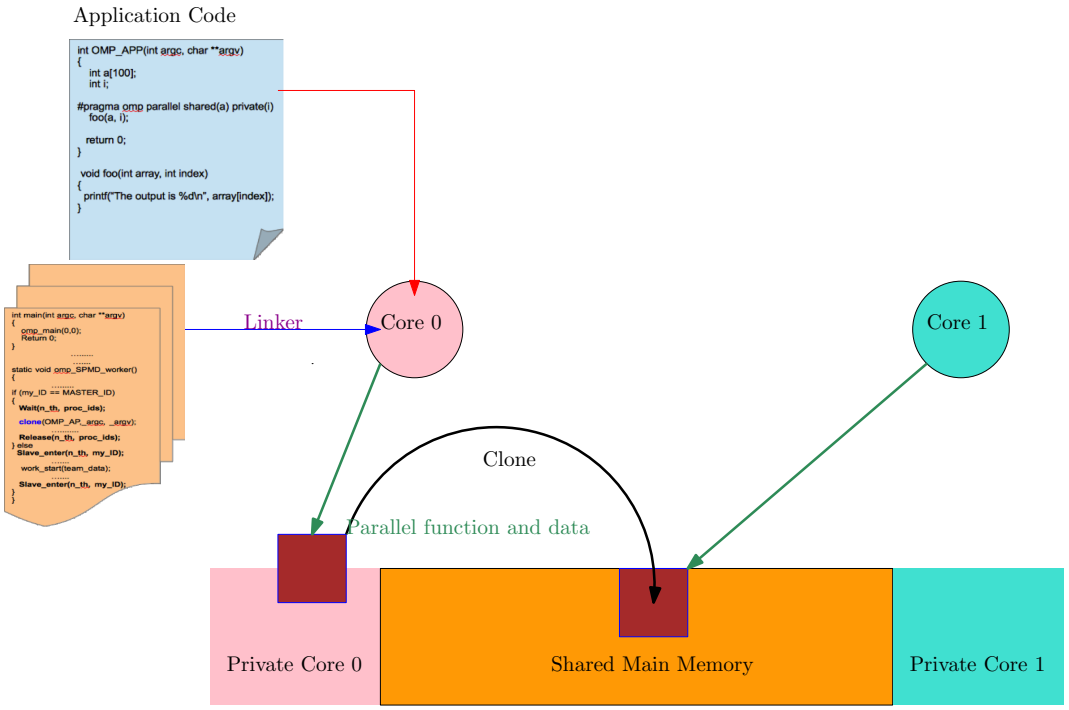


FIGURE 4.4: Abstract view of shared data supported

1. Reserving some rang in virtual memory space of the core.

2. Mapping those virtual addresses to certain physical addresses by instructing the kernel.

3. Configuring the LUT entries to point to those special physical addresses.

Therefore, any access to this virtual address will be translated to the same physical addresses, but the LUT will redirect the access to the new target.

As explained in Section 4.1.1, the compiler separates out each code portion that belongs to parallel construct and outlines it into a separate function. This outlined function has shared data and also additional parameters such as the lower and the upper bounds of the work-share loops. After the compiler transforms the outlined function, the run-time function will indirectly clone the outlined function in the master core. As shown in Figure 4.4, my implementation is to clone the outline functions in physical shared memory so that the master thread has one copy of the function in its private memory, and one for slaves in shared region. It performs cloning by using the **clone()** system call that provided by Linux [173]. It is a new level of process context creation that needs a pre-allocated memory area is supported by the programmer. So, it can clone the whole sub-graph of an outlined function with its arguments. This implementation avoided all issues were related to deal with auto variables by any parallel region. By this way, it will not need to create a shared stack and copying the value of the original

variable there, then flushing to their original area at the end of the parallel region and freeing the space after the team seizes execution [76, 93, 174].

This way has one limitation is that it can't use the pointer approach to passing the shared data to the slave. Because of the virtual address of master core is not valid for other cores. The current architecture of MPSoC and special the SCC have a set of independent processor cores each running their own operating system kernel. Additionally, there is no part of the system state is shared or synchronized across the chip and the kernels do not know of each other. Now, it cannot freely exchange virtual address across cores.

To overcome this obstacle, it relies on a sort of marshalling operation in GCC that generates *metadata*, as illustrated in Listing 4.2. Here, the pointers are replaced with shared data which are involved in metadata by its offsets relative virtually to the base address of shared region.

```
typedef struct
{
        int [100] *A;
}omp_data_s;
```

Listing 4.2: Compiler genertated Metadata

However, offsets are used alternatively and let each core add the base (virtual) address of the mapping. This idea supported by the Microsoft C compiler that is known as "*_based*" specifier for pointers. Here, pointers behave like normal ones from a programmer's view, but they stored in memory just as offsets, relative to the specific "base" address. Unfortunately, this approach is not supported by the GCC compiler yet.

Therefore, the offset arithmetic is hid behind some compiler syntax. Here, the GCC compiler extended to support the basic "*_based*" idea. Listing 4.3 and Listing 4.4 show an example code and the compiler transformation to explain my novel approach, respectively.

```
int OMP_APP(int argc, char **argv)
{
int a[10], i=0;
#pragma omp parallel shared(a) private(i)
while (i++ != 10)
{
   a[i] = i;
}

return a[i-3];
}
```

Listing 4.3: Example code

In Listing 4.3, *a* is a permanent shared array and will reside in shared memory after the cloning it implicitly. Listing 4.4 shows how the compiler will transform the code on the master (*SENDER*, up) and slave (*RECEIVER*, down) side.

```
/* SENDER Side */
OMP_APP (int argc, char * * argv)
{
  int i;
  int a[10];
  int D.5171;
  int D.5170;
  struct .omp_data_s.0 .omp_data_o.1;
  void * D.5185;
  int[10] * D.5182;
  void * D.5183;

<bb 2>:
  i = 0;
  D.5185 = __builtin_GOMP_compute_sender_offset (&a);
  .omp_data_o.1.a = D.5185;
  __builtin_GOMP_parallel_start (OMP_APP._omp_fn.0, &.omp_data_o.1, 0);
  OMP_APP._omp_fn.0 (&.omp_data_o.1);
  __builtin_GOMP_parallel_end ();
  D.5171 = i + -3;
  D.5170 = a[D.5171];
  return D.5170;

}
;; Function OMP_APP._omp_fn.0 (OMP_APP._omp_fn.0)

/* RECEIVER Side */
OMP_APP._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i)
{
  int a[10] [value-expr: *(int[10] *) __builtin_GOMP_compute_receiver_offset
    (.omp_data_i->a)];
  void * D.5209;
  int[10] * D.5208;
  _Bool D.5207;
  int i;

<bb 8>:

<bb 3>:

<bb 5>:
  D.5207 = i != 10;
  i = i + 1;
  if (D.5207 != 0)
    goto <bb 4>;
  else
    goto <bb 6>;

<bb 6>:
  return;
```

```
<bb 4>:
  D.5208 = .omp_data_i->a;
  D.5209 = __builtin_GOMP_compute_receiver_offset (D.5208);
  MEM[(int[10] *)D.5209][i] = i;
  goto <bb 5>;

}
```

LISTING 4.4: Illustration of compiler transformation of shared data

Here on the SENDER side, the master thread stores the offset of shared variables that is an output of a call to *GOMP_compute_sender_offset(&a)* into omp_metadata, then passes the structure's address to the run-time environment and makes it available to slaves. This function is implemented in the *libgomp_scc* library that returns value is a pointer, which is computed as (*RETURN (&a - SHMEM_BASE_ADDRESS_ON_CALLING_CORE)*), where the *CALLING_CORE* here is the master thread.

On the RECEIVER side, each access to a[...] is translated into an access to a pointer retrieved to the *GOMP_compute_receiver_offset*(omp_data_i− >a). This function also supported by *libgomp_scc* run-time to return value is (*RETURN (*omp_data_i− >a + *SHMEM_BASE_ADDRESS_ON_CALLING_CORE)*), where the *CALLING_CORE* here is the slave thread.

Finally, the compiler traditionally replaces all shared variables within the outlined parallel function with references to the corresponding fields of the metadata structure.

This approach will definitely save on pointer arithmetic and some offset calculations don't carry any weight. In addition, it does not need to change existing routines to perform necessary offset calculations and it avoids carefully check and fixes each memory access to shared variables. In the end, every core could access to those variable by using the original mechanism of the GCC compiler.

4.1.4 Thread Creation and Management

In a traditional OpenMP implementation, *libGOMP* manages a pool of threads [17]. Namely, it can add new threads only when the thread pool is empty and the number of threads is usually much larger than the number of cores on the platform. A thread is added automatically to a thread pool without removing to reuse later at the end of a parallel region. However, this implementation has limitations: threads consume system resources (e.g. stack space), to utilize the thread pool must be managed efficiently to avoid the increasing number of idle threads that affect the runtime performance

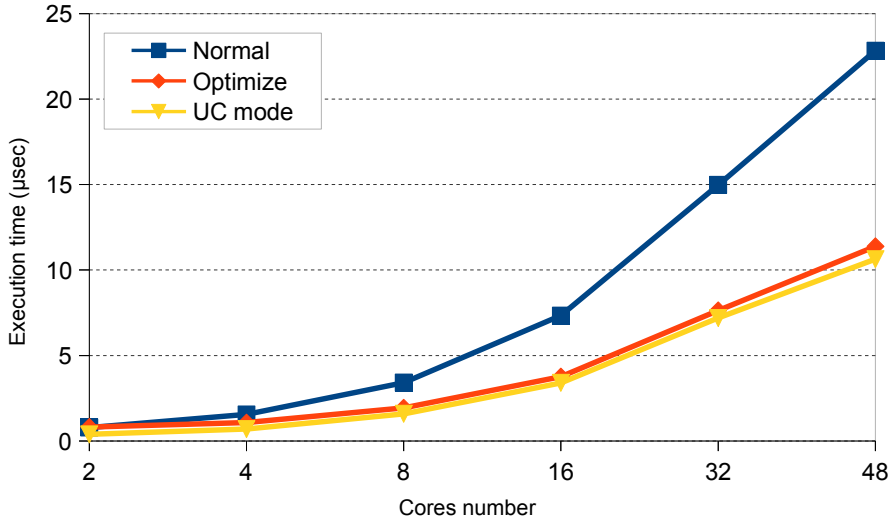


FIGURE 4.5: Impact of the cache alignment on MPB access

significantly. Moreover, P54C core on SCC doesn't support multi thread mechanism and it consequently runs one task at a time.

To handle this situation, **elastic metadata** primitive used to query all of thread-local data associated with thread team. To guarantee no system resources are wasted and to eliminate many of the management problems inherent in the traditional OpenMP implementation. Furthermore, this approach ensures that the spinning task executed by a slave thread (not into parallel region) does not interfere with the sequential parts of the program execution on the master thread.

However, this *metadata* is created when the master thread encounters a parallel region, which holds all necessary information about the work to be executed by slaves. It calls this information as *thread descriptor* that contains two main blocks:

(**a**) *Thread information* that has a pointer to code of parallel function and its arguments.

(**b**) *Control information* that holds a number of threads, array of the local IDs assigned to processors, and the work-share descriptor with its synchronization primitives.

Here, the master thread allocated the *metadata* on the main shared memory as shown in Figure 4.6.

Once the parallel region starts, the master thread updates the *metadatas*, it is storing the address of *metadata* in a global *TEM_DESC_PTR* array (each location assigns one core). To address the issues of interference traffic on the interconnect and atomic access, multi copy of *metadata* pointer used and distributed over *TEM_DESC_PTR* array. *TEM_DESC_PTR* resides on the local memory (MPB) of the master thread to avoid

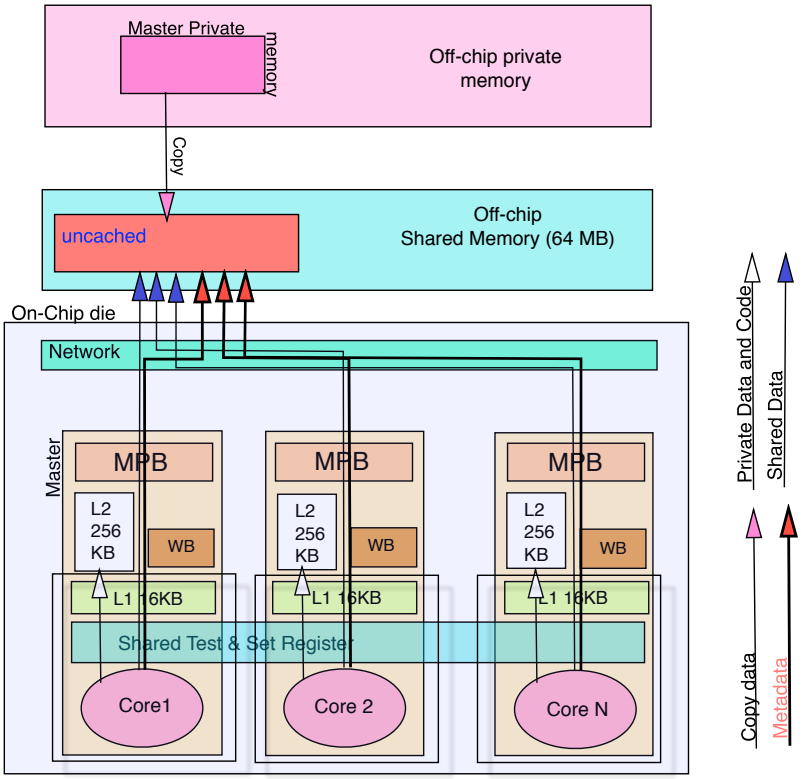


FIGURE 4.6: Data and metadata allocation

the slow access to off-chip and the extra overhead of *aligning the data allocated in MPB with cache lines* as depicted in Figure 4.5.

Figure 4.5 shows the results of the cache alignment on access to MPB in which an increasing number of cores. It implemented this micro-benchmark for read and write concurrently from/to MPBs (under the same setting in the Section 3.1). In this micro-benchmark, the overhead (μsec) measured by: (i) one loop (Normal) is responsible to read one byte allocated on one MPB (MPBT mode) and to write it again to same address in every iteration, (ii) two loops (Optimize) are used to read and to write one byte separately to the same address in the same MPB. As explained in the figure, *Optimize* implementation has overhead less than 50% (for 48 threads) compared to *Normal* implementation in MPBT mode. While, the impact of separating loop into two has added little bit overhead (resulting from adding an extra loop) when the number of cores up to 2, Otherwise, it did not find such evident optimization impact in the performance of the same *Normal* implementation executed in *un-cached* mode (*UC mode*). Hence, every access should fall on the same cache set in each loop. Therefore, the *TEM_DESC_PTR* allocated in the master's MPB by using MPBT mode as a baseline and also because of this memory is close to master core and it has intensively access to them.

However, all slave threads can join the parallel region by access their *metadata* based on the core id. This approach relaxes the condition of identical threads implementation

to allow each thread to be distinct (heterogeneous threads) by relaying on core ids. It also supports nested parallelism as well by porting special *metadata* that contains multiple threads (team).

4.1.5 Synchronization Primitives

The OpenMP has a number of means to perform synchronization between parallel threads, such as *critical*, *atomic*, and *barrier* [34]. The critical construct restricts a region of code to be executed atomically by a single thread. Whereas the atomic construct provides atomically access to a single memory location. The OpenMP compiler translates the beginning and the end of a critical/atomic section to library functions which acquire/release a lock. These directives supported in the original OpenMP runtime library based on a two-level lock: mutex and spin_lock, where the POSIX thread library provides the first one. However, to handle *critical* and *atomic* functions, it can use the synchronization hardware available (i.e, *test-and-set* registers) in the SCC.

Barriers –implicit or explicit – are central constructs to the OpenMP parallelism model and to any shared memory program. Implicit barriers are using usually at the end of parallel regions to ensure that the slave thread does not start until all threads have completed the first parallel work. Explicit barriers (`#pragma omp barrier`) may use by the program developer to ensure all threads arrive this synchronization point, even if arriving at different times due to different workloads. The *fork/join* model uses two synchronization events per parallel loop. Consequently, the costs of barrier deserves more attention, especially in case of the nested loops in an application such as parallel inner and sequential outer loops. Therefore, the overhead of the barrier was recognized as an important source of the performance degradation in the parallel programs [175–177]. Ordinarily, Compiler-generated parallel code may include more barriers than necessary, so it is important to reduce the cost of a single barrier operation to a minimum.

Several implementations of OpenMP for MPSoCs have adopted a centralized shared barrier [41–43]. The centralized shared barrier relies atomically on shared entry and exit counters through lock-protected write operations. The counters are using to hold the number of threads that reached a barrier. The last thread arrives the barrier, it is signalling the waited threads by setting the flag. Synchronized access of the different threads to the shared counter is done using mutex. This algorithm yields bad performance as the access to the counter is *serialized* [105].

In addition, in non-cache coherent systems such as SCC, the barrier structures (e.g. control flags, counters) in shared memory must explicitly be kept consistent with the updates. To work around this problem, it considered several barrier algorithms which

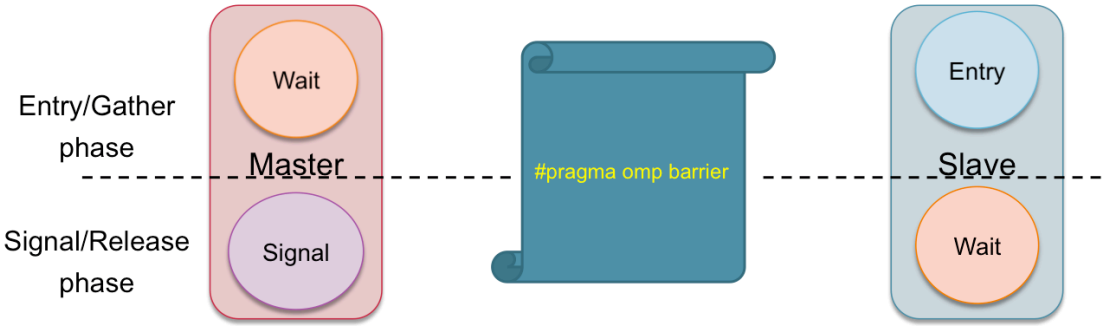


FIGURE 4.7: Master/Slave Approach

leverage specific hardware support for synchronization, different communication pattern, or its explicitly-used portion of the memory hierarchy [178–180].

All barrier algorithms are implemented based on the *busy-waiting* approach. This approach prevents all participants from leaving the barrier point until all threads have reached it. Typically only one thread, the *Master* knows whether or not all threads have arrived at the barrier. Then, this Master sends a wake-up signal to release the waiting threads.

In this section, different barrier algorithms are examined to investigate ways in which OpenMP model and its implementation can scale to large thread counts. Before presenting a brief description of the various barrier implementations, I would first like to give an overview of software implementation for barrier algorithms.

4.1.5.1 Software Implementation

Typically barrier synchronization has three phases. A thread first mails its entry into the barrier, then waits for the last thread to arrive at the barrier, and in the end, it receives a notification signal (from the master thread) to release all threads in the barrier. Those phases are implemented as separate functions in order to analyse the overhead individually for each phase.

The *Master/Slave* scheme considered to implement barrier algorithms on the SCC. This approach is accomplished in two phases, the *Entry* or *Gather* phase and the *Signal* or *Release* phase, as explained in Figure 4.7. Each slave signals its entry into the barrier by using a `SLAVE_ENTER()` function; it then waits for the release signal. Master thread controlled Entry and Signal phases by using `WAIT` and `RELEASE` functions (as shown in Figure 4.2). Those functions were implemented individually to execute additional housekeeping functions before releasing the slave threads. Here, the local memory (MPB) used to store the according flags. These memory spaces are fast and shareable

among all cores without any issue in coherence to exchange vitality messages between the individual SCC cores.

The SCC provides two kinds of memory access usages to the MPB, *cached* (MPBT) and *un-cached* (UC). In MPBT mode, the data caches in the (L1) cache only as explained in Section 3.1. While in UC mode, data reads and writes are directly issued to the network. It is possible to mix UC and MPBT modes for accessing the same physical address in MPB as illustrated in [120].

The barrier algorithms were implemented by exploiting shared-bytes and MPBT memory mode. A common implementation for a read-update-write operation by using MPBT mode is depicted in Listing 4.5. Firstly, the *CL1INVMB* invalidates all L1 cache lines of MPBT type before the subsequent read access in order to get new updated values from MPB; likewise, before write accesses it enforces writing updated data towards the Write Combine Buffer (WCB).

```
CL1INVMB();
<read byte(s)>
CL1INVMB();
<write modified byte(s)>
FLUSH_MPB();
```

Listing 4.5: MPBT mode

Then WCB is flushed by issuing a second write with a whole cache line at different addresses. By exploiting UC mode (Listing 4.6), it can optimize the performance of barrier algorithms by avoiding extra overhead for invalidating MPBT lines before read and write operations as well as the cycles required to flush the WCB, as illustrated in the previous work [178, 179].

```
<read byte(s)>
<write modified byte(s)>
```

Listing 4.6: UC mode

Therefore, it uses MPBT data in my implementation of the barrier algorithms (excluding RCCE implementation) as the baseline method, which allocates the flags in the MPB by using MPBT mode.

4.1.5.2 RCCE algorithm (RCCE-B)

The SCC platform supports the message-passing programming model. The *RCCE* [128] is one well-known library to support this model, featuring a simple barrier algorithm

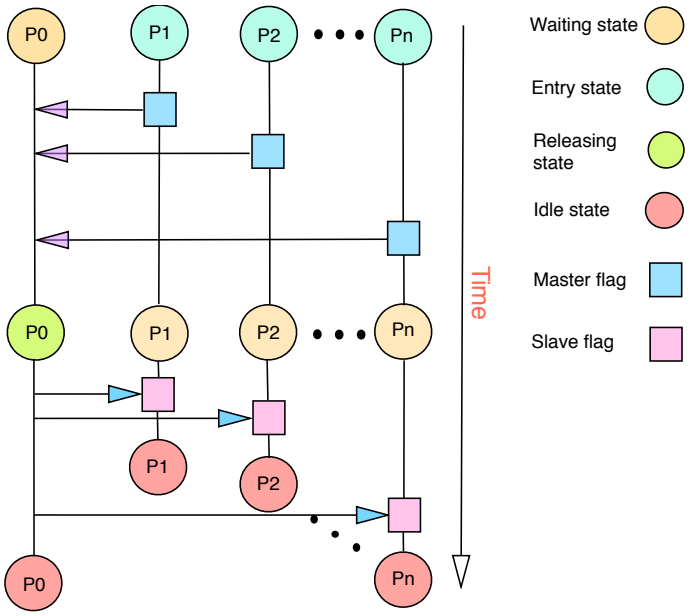


FIGURE 4.8: RCCE Barrier

based on a *local put / remote get* mechanism. It exploits flags for synchronization by allocating them in that core's MPB to initiate an update.

This is the baseline implementation of the barrier algorithm, which allocates an array of flags in the local shared memory (Master's MPB), every core initializes its own flags therein. In this implementation as depicted in Figure 4.8 (where threads are represented by a circle, time flows downward, and a square shape symbolized memory allocation), they used the MPB to allocate flags, since it is shareable between all cores without any coherency issue. Here, a master thread is responsible to gather and release waiting threads. As a consequence, the Master core polls remotely on the release flag repeatedly for all following cores.

4.1.5.3 Shared-Master-Slave algorithm (S-MSB)

It is linear barrier implementation as shown in Figure 4.9. This algorithm is an extended version of the Master-Slave schema [105]. It is extended by Marongiu et al.[45] to use a message passing-like approach for signalling by allocating each of the slave poll flags onto their local memory. To address the issue of the traffic generated by polling activity that is still generated through the interconnect towards shared memory locations, which potentially leading to congestion.

In this algorithm, the master core accepts all the entry signals at the barrier and then it is issuing release signals. As a result, the order of entry signal acceptance is fixed, since every slave sends its status by using a separate flag. After the notification step, each slave

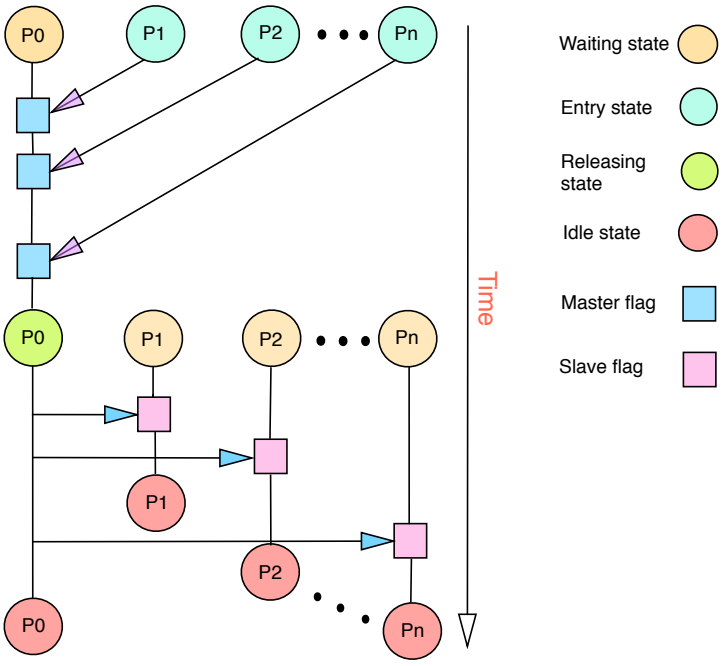


FIGURE 4.9: Shared Master-Slave Barrier

enters a waiting state and polls on its private location for the release flag. In the signal phase, the master thread updates each slave's polling flag by broadcasting a release signal. The S-MSB approach removes contention for shared counters and congestion by distributing each of the slave's poll flags in their local memory (MPB). Every slave core is responsible to initialize its own flag that allocated in the Master's MPB and the Master core polls flags therein. This scheme uses a *local get/remote put* approach as depicted in Figure 4.9. Two flag arrays are used in different memory portions: Master's MPB contained Master flags and Slave flags are distributed over the Slave's MPBs. This algorithm is presented in my previous work [178]. It optimized the performance of this algorithm by separating the loop access to Master's flags into two loops, as it can see in the next section.

The two loop implementation used to optimize the performance of this algorithm through separating the loop access to Master's flags to reduce the cache misses. Here, the gather loop of Master flags in `WAIT()` function is separated to two loops: one is gathering the slaves' entry signals, the other is reinitializing them. Consequently, it has influence by approximately the same performance ratio of the `SlAVE_ENTER()` function in the slaves as explained in Section 4.3.1. This impact is the result of *aligning the flag allocation in MPB with cache lines*, where every access in each loop is falling on the same cache set.

FIGURE 4.10: Time measurement in barrier algorithm

4.2 Methodology and Micro-benchmarks

4.2.1 Barrier Implementation

An important factor in determining the performance of a shared memory system is the overhead due to synchronization for language constructs in OpenMP. Furthermore, the costs of these operations are dependent on their implementation in the OpenMP runtime library. To find out what the cost for each construct is, it needs to analysis the overhead that associated with barrier phase implementation.

TABLE 4.1: The parameters of the barrier performance model

Parameter	Description
O_g	Core overhead to read the flags in Master thread.
L_g	Communication time to gather flags in Master thread.
O_c	Core overhead to check the status of flags.
O_s	Core overhead to update the flags in Master thread.
L_s	Communication time to update flags in Master thread.
O_e	Core overhead to update the flags in Slave thread.
L_e	Communication time to update flags in Slave thread.
L_r	Communication time to read flags in Slave thread.
O_r	Core overhead to get the new flag signal in Slave thread.

However, Culler et al. [181] proposed *LogP-model* that accurately predicts performance of a complex program on active-message based systems. This model has four parameters to summarize the performance of a platform: the network latency L, the overhead O, gap g represents the minimum time interval between subsequent messages, and the processor number P. It uses this model to represent the overhead of barrier algorithms as depicted in Figure 4.10. The parameters of this model are listed in the Table 4.1. A simple micro-benchmark used in most previous studies to estimate the time only in the Master core. The latency of gathering and releasing the participating processors only is measured as illustrated in the Figure 4.10 (red bar). The overhead of slave threads in the phases of the barrier was overlooked by the researchers. In Figure 4.10, a traditional way to measure the overhead in barrier algorithms by padding the time read before and after the barrier algorithms in the Master thread (red bar). As a consequence, missing the time of travail signal (yellow bar), response time of flag change (blue bar and part of waiting time in pink bar), and the time of release (green bar). Therefore, it can classify the overhead into two sites, Master and Slaves try covering all the time consumption. This method, to best of my acknowledges has never been used before in the performance measurement.

However, it can calculate overhead of LogP-model parameters (Table 4.1) by measuring the time in two groups which are *Master Overhead* (MO) and *Average Slave Overhead* (ASO). The MO introduces the cost for of performing barrier synchronization in the Master thread, including the two barrier phases. The overhead of inserting the new value of flag (yellow bar in the Figure 4.10) is including in ASO site. The ASO gathers the overhead per participant (excluding the Master) by summing up all slaves' overhead divided by the number of participating slaves. As a result, this procedure gives a direct comparison of barrier costs on Master and Slave and therefore estimating the quality of the evaluated algorithms based on the execution time and parallel speed-up. To best of my acknowledge, this method has never been used before in the performance measurement.

The Edinburgh Parallel Computing Centre (*EPCC*) represents a simple micro-benchmark [182] used to measure barrier performance in OpenMP model; however, as it does not estimate the implicit barriers in the end of OpenMP directives, it therefore considers insufficient.

However, the experiments have been carried out by executing barrier code only on the platform: *Pure Overhead*. In the *P*ure Overhead micro-benchmark, barrier code is executed only on the system without any communication between cores takes place. This allows to estimate how the algorithm scales with increasing synchronization traffic only,

as illustrated in the previous work[178–180]. For computing the average pure overhead of the barrier, it uses the following equations:

$$Avg_{MO} = \frac{TotalBarrierTime}{(No.ofIterations - No.ofIgnores)} \tag{4.1}$$

$$Avg_{SO} = \sum_{i \neq Master}^{No.ofthreads} (AvgBarrierTime) \tag{4.2}$$

$$O_P(CPUcycles) = \begin{cases} Avg_{MO} & \text{if } thread \text{ is Master} \\ \frac{Avg_{SO}}{(No.ofthreads)-1} & \text{else} \end{cases} \tag{4.3}$$

Where:

- $TotalBarrierTime$: is the time of the barrier phases without any load overhead.
- Avg_{MO}: is the average time of the Master thread, including `Wait()` and `Release()` functions.
- Avg_{SO}: is the average time of the Slave threads, only including the `Slave_Enter()` function.
- $O_P(CPUcycles)$: is the average execution time of the Master or Slaves in CPU cycles.
- $No.ofthreads$: is the number of running threads (participants).

4.2.2 Parallelism Model

OpenMP is a parallel programming language system that is designed based on the model of fork/join parallelism and the notion of *parallel regions* where computational work is shared among a team of threads. In flat implementation, the fork-join parallel programming model was primarily designed for uniform access shared-memory space (*UMA*) systems and for relatively modest thread counts. Therefore, the overhead for spawning and joining parallelism is an important factor that affects the performance. These overheads, however, become significant for large core numbers.

To better understand and identify overhead of a fork/join model on OpenMP, Figure 4.11 shows a diagram of operation in the OpenMP parallel region. This figure depicts a time-line (gray bar) of the subtasks of the beginning and end runtime code of a parallel

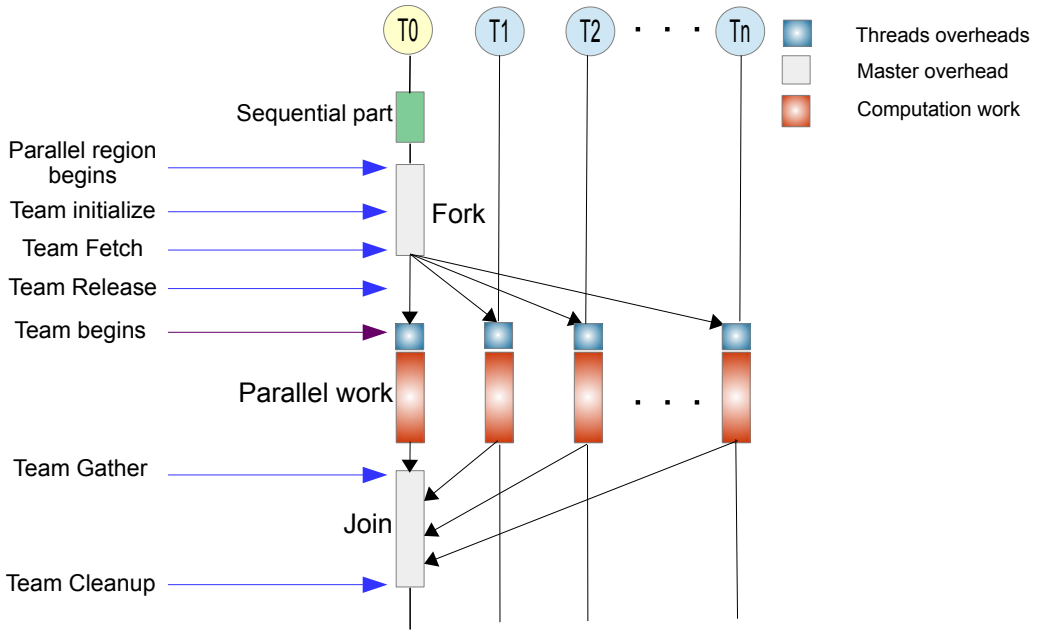


FIGURE 4.11: OpenMP Parallel Region Operation

region construct. Consequentially, the breakdown of fork execution times divided into three main phases:

- **Team INIT**: allocate and populate the team descriptor.
- **Team FETCH**: that used to fetch the slave threads from the global thread pool.
- **RELEASE**: release the slaves from global synchronization structures.

Similarly, the breakdown of join execution times plotted into the main phases: the time to collect the team threads on the synchronization structure (**GATHER**), and the time to tear down the team descriptor (**CLEANUP**).

Based on the Figure 4.11, the EPCC micro-benchmark [182] is extended for capturing the overhead of parallel OpenMP events. Conventionally, the EPCC measured the overhead of OpenMP directives by comparing the time taken to the code section executed sequentially by the time taken for the same code in parallel mode, including the costs for creating parallelism and synchronization primitives. As a consequence, this benchmark is appropriate only for single-level parallelism; to use EPCC with a large number of threads (such as in many-core system) can estimate an inaccurate overhead. In addition, evaluating OpenMP parallelism based on application speed-ups (e.g, NAS Parallel Benchmarks (*NPB*) [183] and the Standard Performance Evaluation Corporation (*SPEC*) [184]) depicts the overall performance indications without revealing potential construct-specific problems.

TABLE 4.2: OpenMP directive transformations

Before	After

/* Parallel Region */
 	
#pragma omp parallel
Work_load();
	getclock();
	GOMP_parallel_start(&omp_fun, &omp_data, num_threads);
	omp_fun(&omp_data);
	GOMP_parallel_end();
	getclock();

GOMP_parallel_start(&omp_fun, &omp_data, num_threads);	*gomp_team_start(&omp_fun, &omp_data, num_threads,* ***Team_INIT****,* ***Team_FETCH****);* **getclock();** *RELEASE();* **getclock();**
GOMP_parallel_end();	**getclock();** *GATHER();* **getclock();** *gomp_team_end();* /* CLEANUP */ **getclock();**

Therefore, Table 4.2 shows my proposed extension and performance library routines which can be used to monitor an OpenMP event. It integrated a special instrument inside the OpenMP run-time library based on the directive transformations for specific purposes. As a result, this way allows me, first, avoiding extra overhead that caused by the technique of directive transformations, and second, it shows more performance details about interesting OpenMP execution events (e.g., team creation, fetching threads, thread synchronization, clean-up). Finally, this way can be easily accommodated several measurement modes such as profiling [185, 186] and tracing [187, 188] without compiler involvement.

In Section 4.3.2, two different implementations of micro-benchmark used to compare the performance and scalability based on the above scenarios: *Impact of static load*, and *load imbalance*. The static load implemented by adding the same work load to every thread included the master thread. By this way, it will be sure that each core arrives at the end of the parallel region with maximum efficiency.

In most applications, the load imbalance poses a challenge to achieving satisfactory parallel efficiency. Therefore, it exploited the load imbalance approach to evaluate the overhead of initializing, creation, and terminating parallel threads. It implements a synthetic micro-benchmark that is a function. This function will perform several floating point operations without any data involved. Where, each core in the parallel region will call the number of times this function which will be represented the load of each OpenMP thread. Excluding one core (Master thread) that uses a larger number of iterations that is 4x more than in others. As a result, this benchmark helps to understand where time is wasted due to mitigate the overhead of the implicit barrier within the end of the parallel region. By reducing the time of waiting threads to reach the implicit barrier, where the master thread will be sure enter the barrier line after all threads arrival.

4.3 Run-Time Overhead

In this section, It evaluates the overhead cost that imposed over program execution time by *libgomp_scc* services. It is important in many aspects of OpenMP research to understand the challenge of adapting OpenMP to emerge MPSoC platforms. All the experimental evaluation in this section has been generated by using the defaults SCC settings which mentioned in Section 3.1. The experiments have been carried out by executing parallel code on different core counts.

In Section 4.3.1, It discussed the cost of each of the phases (gather + release) of the barrier algorithm that can potentially improve the performance. To better understand the overhead of a fork/join model is detailed in Section 4.3.2. Finally, to study the impact of memory access mode on the performance, Section 4.3.3 presents the results of barrier approach; and validated its performance with the original implementation.

Nevertheless, It believes that benchmarks do actually cover most of the characteristics that real applications feature. Thus enabling me to make realistic assumptions about the performance of real applications.

4.3.1 Synchronization Primitives

Figures 4.12 and 4.13 show the performance of the pure overhead micro-benchmark. This section compares the performance of S-MSB barrier and its optimized version (S-MSBO) implementations with RCCE barrier. Figure 4.12 shows the direct comparison of these algorithms. This figure depicts only the cost of the barrier code without any other form of communication between threads as explained in Section 4.2.1, to show

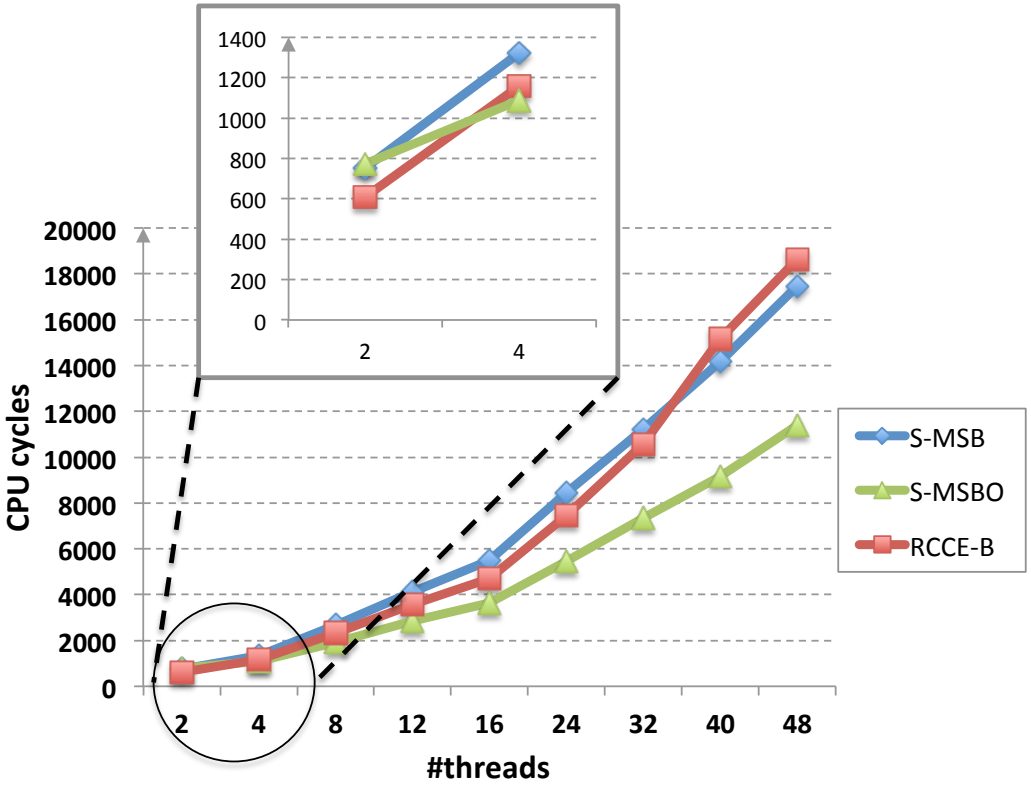


FIGURE 4.12: Pure Overhead of Barrier Algorithms

how the barrier algorithm will scale with increasing traffic by synchronization only. The scalability is analyzed by varying the NoC topology as the number of cores increases. The size of the topology represents only by the core numbers, however the topology takes also into account the external off-chip device.

Obviously, to analyze the Figure 4.12, the implementations of the RCCE-B, S-MSB and S-MSBO are linearly dependent on the number of cores as expected. S-MSBO's overhead is less than 39% and 35% (for 48 threads) compared to RCCE-B and S-MSB respectively. In spite of SCCs capabilities for flexible use of on-die resources for efficient synchronization and communication between cores, the high cost of this barrier algorithm is not surprising. Employing a distributed-shared algorithm allows to completely remove the traffic due to busy-waiting as compared to RCCE barrier. The S-MSB scheme, as expected, mitigates the effects of the bottleneck due to contended resources for core numbers > 32.

Regardless system size and NoC topology with no hardware support, S-MSBO shows always the fastest barrier, making it the ideal candidate to perform synchronization. For more than two cores, S-MSBO shows significantly speed-up more than others. This impact is the result of *aligning the flag allocation in local memory (MPB) with cache line*. In addition, the two loop mechanisms (*loop fission*) also support better utilization

of *Write Combine Buffer* (WCB), since WCB collects all bytes and sends them as a single batch. Using 2 cores, S-MSBO adds more overhead that is resulted of two loops accessing the same memory bank as shown in Figure 4.12.

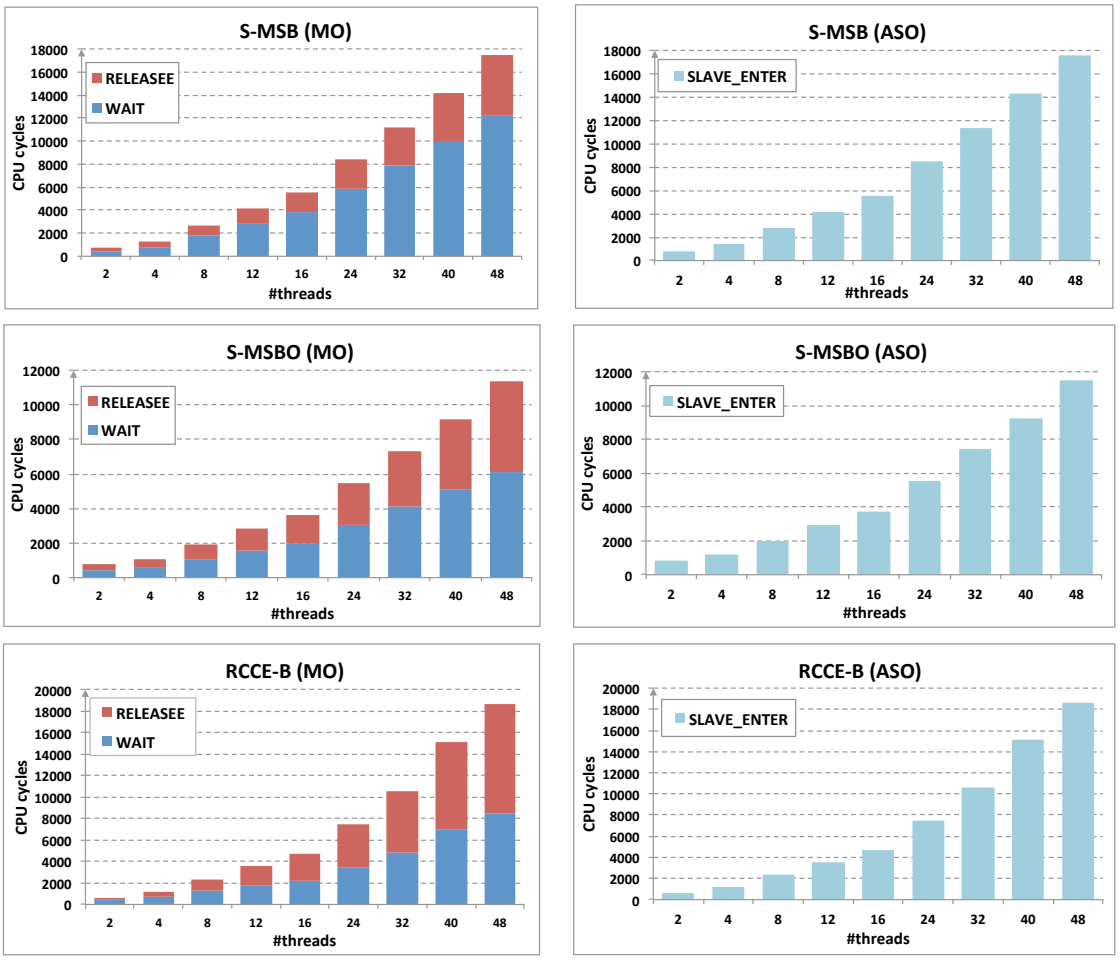
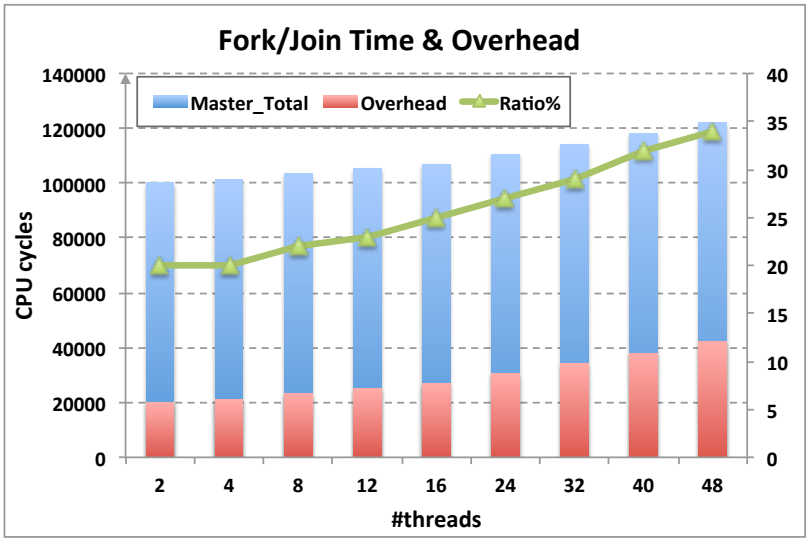


FIGURE 4.13: Pure Overhead of Barrier phases (Gather & Release)

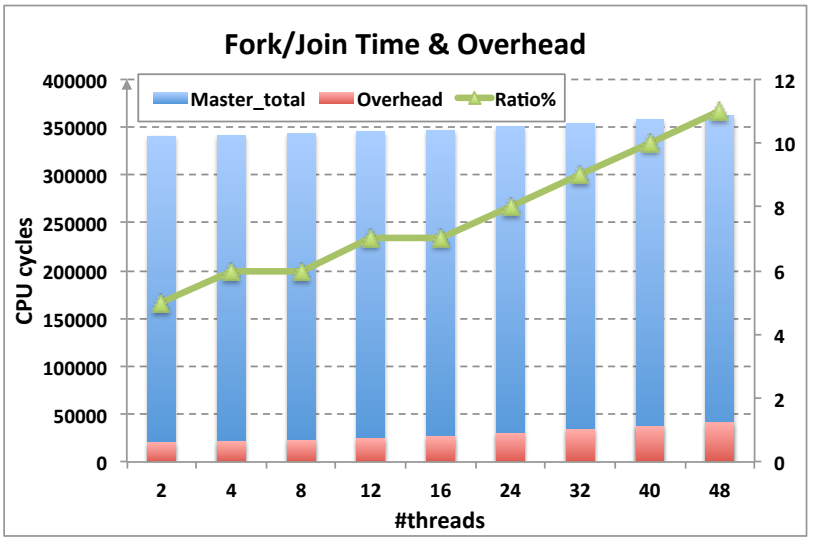
As a consequence, S-MSBO mitigates the effects of the bottleneck due to contended on-chip resources and it used therefore as a baseline implementation, which allocates the flags in the MPB by using MPBT mode. The cost for each of the barrier phases gather and release plotted into MO and ASO, respectively in Figure 4.13. The first column in Figure 4.13 shows the average overhead of the Master thread that includes waiting-time and releasing-time of the slaves. While the second column depicts the average overhead to inform the Master thread by the slave and waiting-time for the exit (release) signals. As depicted in Figure 4.13, RELEASE() and WAIT() functions of S-MSBO in MO achieve more than 48% and 28% overhead reduction compared to the baseline RCCE for 48 threads, respectively. Consequentially, allocating barrier flags in the local shared memory (MPB) of the master thread allows to completely remove the traffic due to bus-waiting and extra communication costs to access to the slaves poll flag.

For the ASO, this overhead reduction as well is reflected by approximately the same performance ration of the SLAVE_ENTER(), because there is no contention to signal the Master thread or to access a shared signal variable. The cost for synchronizing 48 cores reduced to ≈ 11000 cycles.

4.3.2 Fork/Join Model



(a) Static Load parallelism



(b) Imbalance Load parallelism

FIGURE 4.14: OpenMP parallel Overhead

Here, it measures the fork/join cost for the *flat* implementation correspond to a traditional implementation of the runtime of the OpenMP model on the SCC platform. The fork/join mechanism is the responsibility of the master thread. Libgomp_scc library implements this functionality by creating the threads (equal to number that determined by

runtime parameters) and executing the parallel function on all the cores. The extended version of EPCC (Section 4.2.2) measures the overheads for `GOMP_parallel_start` and `GOMP_parallel_end` in clock cycles. Further, it subtracts the execution time of the load essentially yielding the pure overhead for these functions included the overhead of `getclock()` call as shown in Table 4.2. Figure 4.14 explains the total overhead encountered by these functions and its ratio percentage of the total execution time for static and imbalance load benchmarks, while Figure 4.15 shows the distribution of the total overhead over the fork/join execution times. Specifically, the figure depicts two bars correspondence to different number of cores. The blue bar represents the total execution time included work load time and red bar depicts the overhead of fork/join functions.

From the results it is observed that, the ratio of the overhead grows with the number of cores. This is due to the extra overhead of including more threads in the parallel team as it will see later in Figure 4.15. The overheads of fork/join implementation are reduced by 70% approximately when using imbalance load benchmark as shown in Figure 4.14(b). The overhead of the synchronization primitives are as good as for increasing core numbers and the work load while higher for static and fine grained load distribution and creation of parallel regions. The reason lies in the mechanism of the GCC compiler generated code and how parallel regions are handled by the run-time library.

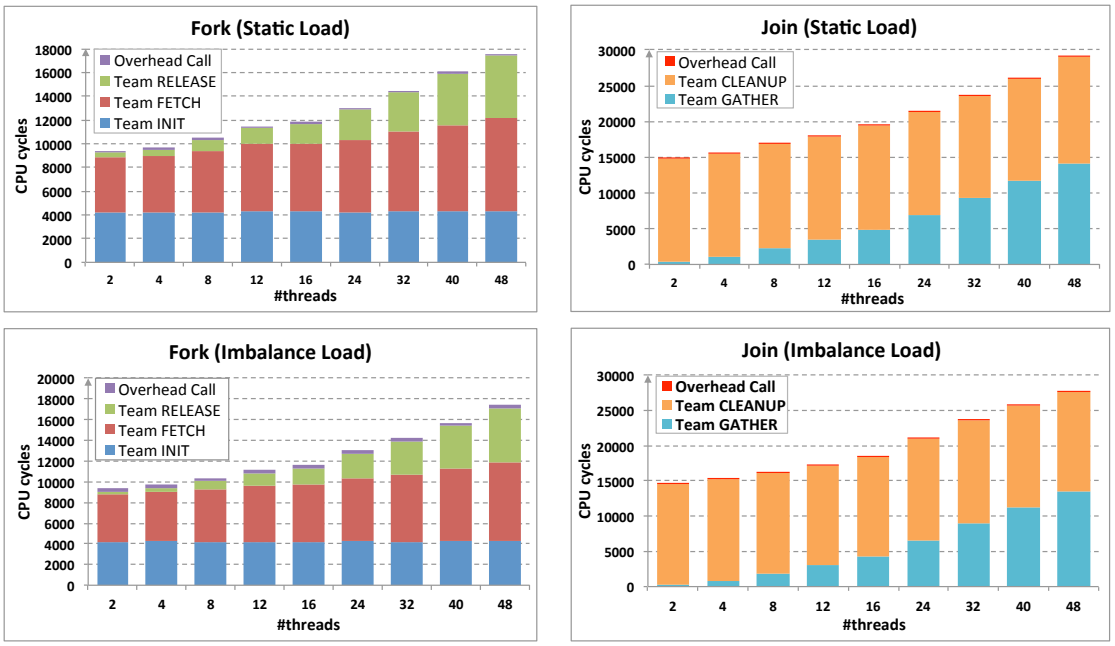


FIGURE 4.15: Cost of Flat fork/join model

To study how efficiently fork/join implementation support OpenMP parallelism, it has plotted the breakdown of both the fork and the join execution times into five main phases as mentioned in Section 4.2.2. Figure 4.15 illustrates the overhead that executed

sequentially by the master thread on the SCC platform. Clearly, in the fork (static and imbalance load), the cost of team FETCH and RELEASE is strongly dependent on the number of threads forked or joined, inasmuch as the extensive communication required among the core that encounters the parallel region, MPB, and the remaining cores. The first component, on the contrary, does not depend on the number of threads requested. The overhead call represents the time spent invoking function for *getclock()* before and after the primitives for fork and join.

Similarly, threads on the dock is collected by iterating over the team participants, so the execution time of this section increased with the thread numbers in the team. CLEANUP, on the contrary, is independent of the number of threads.

Overall, the flat fork/join cost increased with the thread numbers is globally less visible due to a huge time for the INIT, FETCH (during fork) and CLEANUP (during join) stages. Because the data structures were allocated on the main shared memory, which has a very high cost access compared to the on-chip memory on SCC clusters. This cost can reduce by using a custom `malloc` routines, which depend on pre-allocated memory bins with fast inspection. In particular, this cost increased during team FETCH due to the sequential recruitment of a very large of threads on the master. Likewise, during team RELEASE/GATHER (based on S-MSB algorithm), this cost increased because of the NUMA effects that significantly grow remote master-to-slave communication.

4.3.3 Impact of Memory Access Mode

Figure 4.16 shows the timings obtained from the implementation of S-MSB barrier based on UC mode (S-MSBO-UC) on the SCC platform. It also compares this timing in clock cycles with native RCCE implementation, S-MSB, and S-MSBO.

This barrier algorithm that is programmed based on UC mode, it avoids the extra overhead for invalidate MPBT lines before read and write and also the cycles which exploited to flush WCB. The S-MSB-UC implementation clearly improves the S-MSB algorithm, also it significantly reduces the overhead on MPB-based algorithms, as shown in Figure 4.16. It shows worse results by more than 46.6% overhead reduction for 48 threads compared to RCCE overhead in the previous experiments. Obviously, it misses the impact of aligning the flag allocation in MPB with cache lines in S-MSBO performance when it allocated the flags in UC mode. Where, S-MSB-UC shows that implementation benefits from using UC and reduces the overhead by 12.6% for 48 threads.

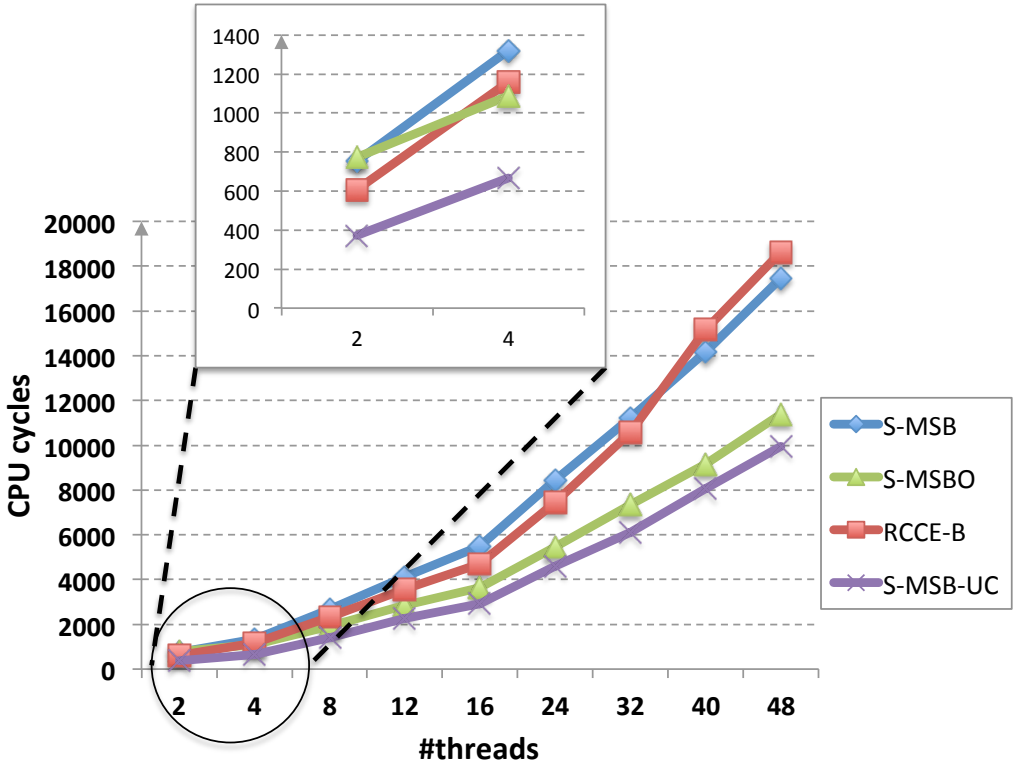


FIGURE 4.16: Pure Overhead of Barrier Algorithms with memory mode

Overall, the choice of a good barrier implementation can dependent on the memory access mode and S-MSB-UC based on UC mode is the best among all the barrier implementations.

4.4 Summary

This chapter talked about the main challenges in the OpenMP design on top of cluster-based many-core system and how to overcome on it. Especially, when the platform that used has no any support to handle the shared data or keep the data coherent between cores. Furthermore, it is attractive to support OpenMP programming model on such system as the SCC. In like manner, the ongoing efforts towards an efficient OpenMP implementation for Intel SCC were presented based on a modified GCC 2.6 compiler with a custom run-time library. This chapter is part in the process of supporting full-OpenMP parallelism to increase programmer productivity, to reduce the design/development costs for the future many-core systems.

Here, the issues and requirements are discussed to support the OpenMP fork/join execution model on the Intel's SCC. In that sense, it believes that significant improvements can be achieved with the mindful usage of relevant architectural features. One of

the keys that used to reduce the run-time overheads is an efficient barrier implementation, because my OpenMP design approach relies heavily on the barrier operations to control threads in parallel. Therefore, it has implemented optimised barrier approach to optimize the OpenMP run-time library performance by considering the impact of cache line with the flag location in the SCC architecture. As part of a quantitative performance evaluation, the experimental results highlight that S-MSBO can obtain a significant reduction in the overhead for the barrier. The S-MSBO allows 48% (MO in Pure Overhead) reduction in the overhead than S-MSB implementation. Consequently, these represent a substantial decrease in the cost of managing parallelism. Furthermore, the memory access mode considered in the barrier algorithm implementation. In the next chapter, more optimization techniques in barrier implementation are introducing.

However, the biggest challenge is data sharing on the SCC to support OpenMP: making shared data from main memory visible to all threads in existence of several OS instances. Namely, each core has different virtual memory space. This chapter handled this issue by proposing a novel mechanism that extended the GCC to share the data and ensuring a consistent view of shared memory in the absence of dedicated hardware cache coherence support. In addition, this chapter shown the preliminary results of fork/join overhead by developing the evaluation criterion with a micro-benchmark to identify the worst cases of the execution time in the model.

Overall, this chapter produced an efficient translation technique to deal with all OpenMP directives with a customized run-time library that responds to the programmer hints for cluster-based many-core template.

Chapter 5

Achieving Low Overhead of Barrier Synchronization Algorithms

OpenMP (and most related shared memory-based programming models) relies on a *fork/join* execution model, which leverages a *barrier* construct to synchronize parallel threads as shown in Figure 4.1. Barriers – implicit or explicit – are central constructs to the OpenMPb execution model and to any shared memory parallel program. Here, the *barrier* directive, `#pragma omp barrier`, ensures all threads pass this synchronization point at the same time, regardless of arriving at different times e.g. due to different workloads. Barriers are implicitly required at the end of any parallel region and work-sharing construct. The unavoidable synchronization overhead has been recognized as an important source of performance degradation regarding parallel program execution. Besides steadily increasing number of threads for parallel regions and the complexity of memory hierarchies in the system, the scalability of the implementations becomes increasingly important.

With the longer-term goal of bringing the OpenMP programming model to SoCs, in this chapter, it is trying to surround the largest possible number of implementations of OpenMP-like barrier synchronization algorithms for SCC. It aims by this work to gain insight into the behavior of different barrier approaches in the OpenMP context in order to determine which of them is a most suitable implementation based on the underlying hardware architecture and number of threads. Moreover, the SCC does not offer hardware cache coherency, arising further problems for shared memory programming models as it is not possible to update a shared memory location by using an atomic operation.

To support atomic operations on SCC, it needs to process a small set of hardware registers, namely *test-and-set* and *Atomic Increment Counters* [189]. Using such hardware primitives eases the construction of efficient synchronization operations in software and according high-level routines built on top.

It therefore investigates a number of techniques for reducing the barrier overhead by considering particular barrier optimizations leveraging SCC-specific hardware support for synchronization and its explicitly-managed portion of the memory hierarchy (i.e., MPB), and pattern communication analysis similar to that performed for message-passing machines. It implemented several approaches of barrier operations integrated into the OpenMP runtime library. The experimental results section provides a detailed evaluation of the performance achieved by different approaches and shows benefits and drawbacks of individual approaches as well as significant performance improvements for the optimal solutions.

5.1 Motivation

OpenMP [16] employs the *fork/join* programming model as depicted in Figure 4.1. The program executes sequentially within a single thread, referred to as the *Master thread*, until it encounters a `#pragma omp parallel` directive. Here, execution forks into a multitude of threads by assigning (forking) computation to a number of worker threads (*slaves*). As a result, a parallel region is created. At the end of the parallel construct the master waits for all slave threads to complete (join) before continuing execution. Then only the Master thread resumes execution. A *barrier* is used to ensure all slave threads have completed before the master thread can continue that is defined as [16]:

"*A synchronisation point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point.*"

One common way to implement parallel regions is to create new threads on the fly relying on standard threading libraries such as *Pthreads*. However, *Pthreads* on SCC would require dedicated abstraction layers to allow threads on different cores to communicate. Unfortunately, the context switch overhead of the Pthread library is rather high as is the scheduling policy of GNU/Linux [170]. Therefore, my approach relies on a custom micro-kernel code [45, 105] executed by every core at start-up. To minimize the cost associated to dynamic thread creation, it assumes a fixed allocation of the Master and Slave threads to the processors.

However, using the *fork/join* model is easy and flexible to handle sequential and parallel parts of an application. It exploits two synchronization events for every parallel

block (e,g., loop, work-share). Consequently, the costs of barrier for *fork/join* model can be high, especially when the application has nested loops such as parallel inner and sequential outer loops.

Therefore, a variety of other barrier implementations exist, which differ in overhead, network traffic, and memory usage [41–43, 175–180]. In order to compare these different implementations on *Cluster-on-Chip* architectures like the SCC, it classified them based on the type of (a) communication pattern employed within the barrier algorithm phases and (b) use of hardware primitives, as described below.

5.2 Linear Algorithms

A linear approach works by a single master being responsible for accepting all the entry signals and issuing release signals. In this approach, the order of entry signal acceptance is fixed. A second approach with similar performance exists. Here, in contrast, each thread receives the entry signal from its next higher-numbered neighbor, and then send its entry signal to the next lower-numbered neighbor. With each core having one predecessor and one successor, this effectively creates a *chain topology*.

5.2.1 Shared Master/Slave Algorithms (S-MSB)

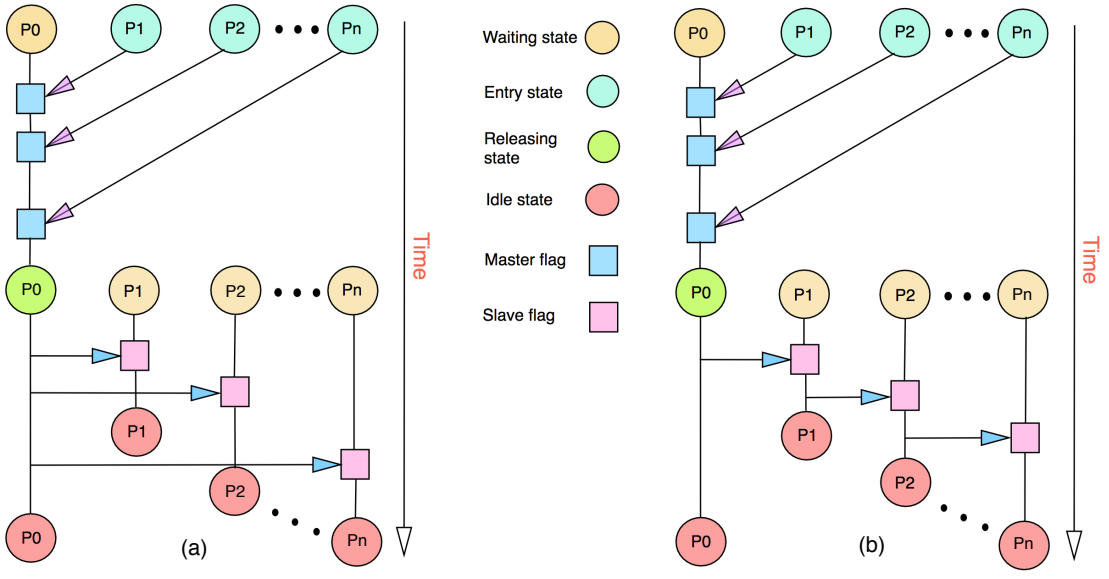


FIGURE 5.1: Shared Master/Slave Barrier

This is the baseline implementation of the barrier algorithm as implemented in Section 4.1.5.3. In this approach, the Master core is responsible for gathering all slaves at

the barrier. Since every slave signals its status using a separate flag, this operation is executed without resource contention. After the notification step, slaves enter a waiting state where they poll on a private location for the release flag. In the signal phase of the barrier, the Master thread broadcasts a release signal to each slave's polling flag. It hence used S-MSBO as a barrier algorithm baseline implementation, which allocates the flags in MPB in MPBT mode.

This scheme uses a *local get/remote put* approach that optimized by by separating the loop access to Master's flags into two loops, as depicted in Figure 5.1 (a). To optimize the Signal phase's performance, it exploited the chain mechanism as depicted in Figure 5.1 (b).

5.2.2 Master Sharing-Slave Algorithms (M-SSB)

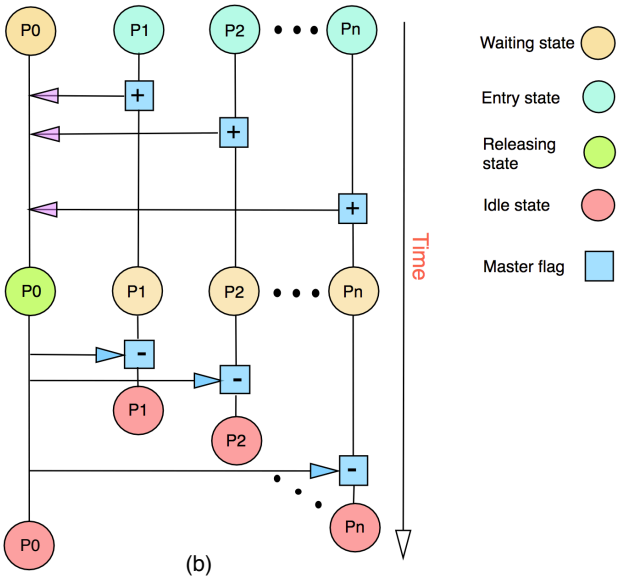


FIGURE 5.2: Master Sharing-Slave Barrier

This algorithm requires one flag array being shared between the cores, as shown in Figure 5.2. In order to notify the Master thread that it has arrived, the Slave thread changes the state of its flag in an array to positive. When the Master thread has verified that this transition has occurred and counts the number of threads have arrived, it releases the Slave by relaying its flag to negative transition. The Slave threads wait until the negative transition occurs. Each core can cause only one of the two transition changes. Since it the MPB is shared between cores and distributed-physically allocation, the access time therefore depends on distance. Here in the M-SSB, every thread is responsible on reinitialize its flag that is allocated in its own local MPB.

5.2.3 Master Polarity-Slave Algorithms (M-PSB)

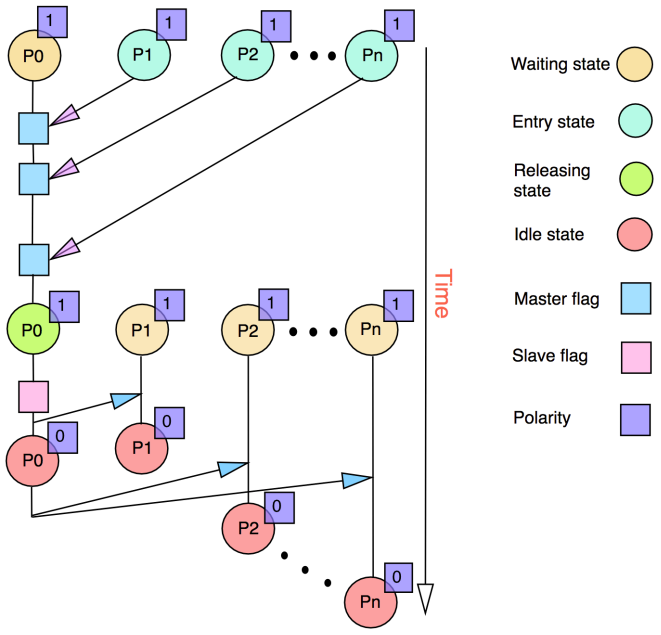


FIGURE 5.3: Master Polarity-Slave Barrier

In this approach, it exploits the polarity exit technique for implementing the Signal phase [190]. This implementation allows me to use a single flag array and one shared variable: in the previous algorithms, the Signal phase of a barrier requires the Master thread release all slaves by broadcasting the *free* signal. By using a single shared variable to convey the release information, it can avoid extra overhead for updating all flags of participating slaves in the barrier. Only the Master thread is responsible for changing the state of this shared variable, the slaves only have read access. The polarity approach is employed for handling reinitialization. It uses a private boolean variable for indicating the current barrier state polarity in order to guarantee correct initialization for derived implementations. Every thread will validate their polarity using a shared release variable. This algorithm shows a possible issue for the case that one thread enters a barrier with a polarity that is opposite to the polarity of another thread. To solve this problem, it starts the barrier using a default initial polarity value. If all threads are entering barrier with same polarity, the above polarity issue does not occur.

Figure 5.3 explains the implementation for this algorithm that is coded by using the linear mechanism for slave gathering. This algorithm provides simultaneous access to a single shared memory location that is limited by the specific hardware architecture; consequently, it generate contention. Furthermore, this algorithm does not give the Master thread any information about the slaves having received the release signal. It only knows when the Slaves enter the barriers and when the Master sends the release

signal. As a consequence, the Master knows that all threads have received the previous release signal once a thread enters the barrier again.

5.2.4 Chain-Polarity Algorithms (CPB)

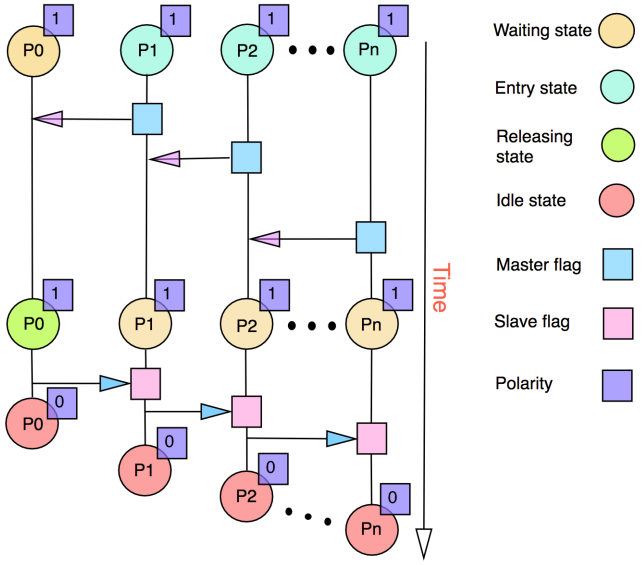


FIGURE 5.4: Chain-Polarity Barrier

CPB in Figure 5.4 is similar to M-PSB(b), but instead of using a single shared variable for releasing the slaves in the Signal phase, this scheme allocates each of the slave's poll flag onto their local memory and uses the chain mechanism to broadcast the release signal [179, 180]. By using this approach, it can avoid contention of the M-PSB algorithms and use the linear approach in Signal phase. The chain mechanism is used in the Figure 5.4 to reduce the overhead of publishing the release signal in the Signal phase. In addition, the polarity approach will not allow to the thread to enter the next barrier with same state of flags in previous one. As a result, the Master thread doesn't need to get notification from last thread in release phase.

5.3 Tree Algorithms

For the following algorithms, it employs the communication pattern depicted in Figure 5.5 for Entry and Signal phases of the barrier algorithm. Yew et al [191] proposed the tree algorithm in order to increase the performance of the Central Barrier algorithm by reducing its associated contention. The tree algorithm exploits logarithmic mapping. It is somewhat similar to the *Master/Slave* scheme. Each phase of the tree barrier approach inherently requires more computation than for a linear algorithm, because each

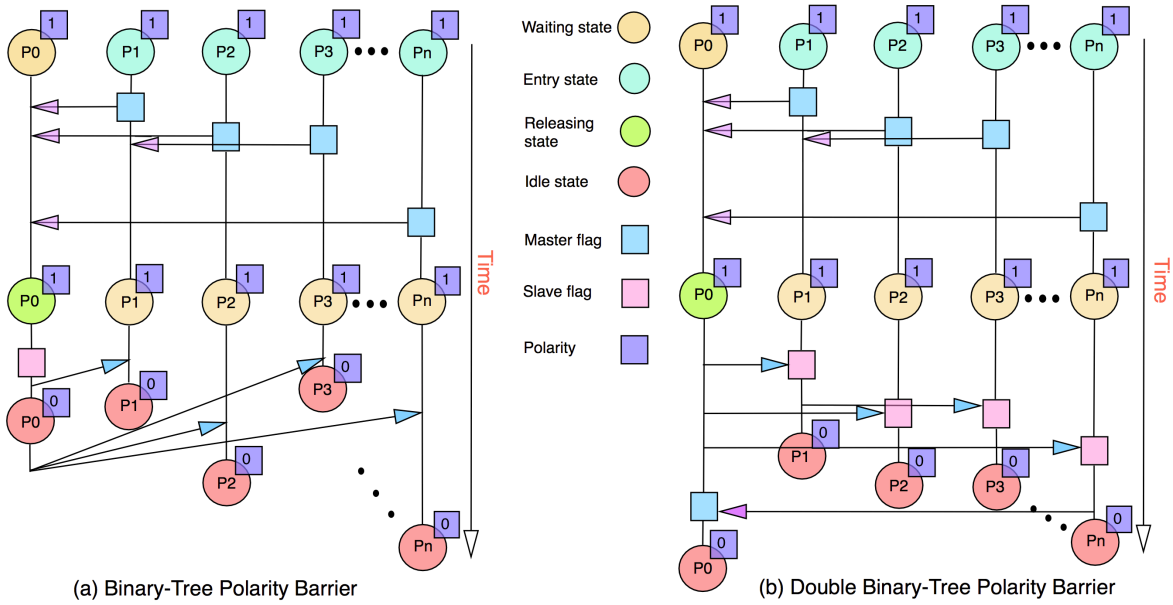


FIGURE 5.5: Tree Barrier Algorithms

thread must calculate its leaves or children. When used for the Signal phase, each thread waits for the release signal from its parent and broadcast it to its children. Despite the slight calculation overhead, it offers strong benefits by reducing the number of transmission steps from $(number\ of\ threads - 1)$ in the case of linear algorithms to $log_2(number\ of\ threads)$ steps.

5.3.1 Binary-Tree Polarity Algorithms (BTPB)

In this approach, each parent node has either two leaves or one leaf, while the leaf nodes are only have a single or no leaves. It allows for a single thread to synchronize with several neighbours and also allows to propagate several signals to others. Traditionally, the binary tree divides the nodes into subgroups (sub-trees) and each node has an array of flags. Every process waits until receiving the entry signal from one higher neighbour and then notifies its own parent. The parent notifies each of its children by exchanging the flag state of the node corresponding to it. The data structures for the tree in each node's parent is initialized to reflect the appropriate child flag that is used to notify the arrival of all children. Then all threads except the root waits for their local wakeup sense flag and the release signal distributed by the parent in each level to its children.

In my implementation, it avoids all this complexity by initializing and allocating an array of flags in every node. To the best of my knowledge, it provides the first implementation of BT-based barrier synchronization on SCC avoiding the complexity issues noted in the previous paragraphs.

Figure 5.5(a) depicts the BT use in the Entry phase. Firstly, it allocates every slave poll flag in the slaves' local memory and also every core is responsible for initializing its own flag therein. Each thread then determines its children according to its assigned node id without the need for knowing its parent. For notifying the parent thread by its children upon arrival at the barrier, each child thread exchanges the state of its flag and waits for the release signal coming from its parent. The parent waits for all children notifications based on individual check-in flag(s) of its child(ren) before updating its own flag; this propagates through the tree until the topmost parent (Master / root) receives the notification. Upon Signal phase, the Master thread sends a release signal to all slaves by updating a shared release value similar to the release mechanism in M-PSB algorithm in Section 5.2.3. The polarity exit scheme is exploited for providing correct initialization of single shared release values for the next barrier iteration. The algorithm therefore needs at most $log_2(number\ of\ threads)$ rounds to complete the barrier entry process [180].

5.3.2 Double Binary-Tree Polarity Barrier Algorithms (D-BTPB)

The D-BTPB is depicted in Figure 5.5(b). Its Entry phase uses the same mechanism as the BT-BP entry phase. Instead of using a single shared release variable for notifying the slaves in the Signal phase, it used also the binary tree to broadcast signals to all slaves. The Master thread sends its release signal to its children once all entry signals of all slaves arrived. On receiving a release signal from its parent, a node forwards it to its children one by one. As a consequence, the broadcast release overhead in the Signal phase is reduced to $log_2(number\ of\ threads)$ steps. To be sure all slaves have already received the release signal, the Master thread waits for the last slave to forward its release signal by exchanging the value of the Master flag that is allocated in the local memory of the Master thread. Thus, the Signal phase overhead in this barrier takes at most $log_2(number\ of\ threads) + 1$ steps, an optimal result.

5.4 Barrier Algorithms using Hardware Primitives

The SCC system has basic synchronization primitives which are implemented in hardware. The synchronization primitives involve Atomic Flag registers (*test-and-set (T&S)*), *AIC*, and *GIR* [123]. In the SCC platform, every core could access those primitives through memory-mapped I/O by using `mmap()` and appropriate LUT entries LUT [127]. Moreover, every core can read and write any of those synchronization registers. Reble et al. in [192] implemented a linear barrier based on the T&S for indicating and releasing threads featuring a lower overhead. Otherwise, using a T&S for each participant

is expensive; in some cases this approach requires mutually exclusive access to shared memory locations, therefore, the resources are scarce.

Petrovic et al. [193] presented a broadcast algorithm based on GIR to address the delay problem using MPB polling for notification. It has exploited this technique in my previous work [178] to reduce the time consumption in the Signal phase of the *Master/Slave* approach by using the user-space library for interrupt handling, but unfortunately the results of overhead were not stimulating. The reason for bad scaling of the interrupt mechanism is *contention*, that is confirmed by Petrovic [193]. Where, there is a fixed set of steps a core should perform when receiving an interrupt. This includes reading from the status register, to determine the sender, and resetting the interrupt by writing to the reset register. Since all the registers related to interrupt handling are on the FPGA, access to them is handled sequentially. Therefore, an interrupt is sent to many cores at once, they all try to access their interrupt status register at the same time, but their requests contend and are handled one after another, which explains the observed performance loss. Consequently, this problem increase the overhead of the barrier.

Reble et al.[192] exploited the a set of atomic T&S registers as flag to indicate and release incoming threads in the barrier schema. Where, each thread performs a linear search for the first unlocked T&S. Then, each thread spins on the T&S, except the last one, enters the release phase by polling on the specific T&S. As a consequence, this implementation has lower overhead compared to the reference implementation of RCCE and minimizes also the contention for access to the MPB. However, it avoids using this implementation because of the numbers of T&S are scarce and the allocation of a T&S for each thread is expensive. In addition, it used T&S to implement atomic and critical directives in the OpenMP.

Reble et al.[189] back to avoid the limitation of usage T&S by exploiting AIC register in Lubachevsky barrier algorithm. The Lubachevsky barrier [194] is a simple algorithm that depends on counters which are visible and can be atomically incremented. On the SCC, it can be implemented using exponential back-off and AICs [189]. This implementation significantly reduces contention and leads to promising results. The counter is used to track the threads: each thread increments a counter to record its presence and then polls that counter to determine the number of threads arrived. As a consequence, the last thread is responsible for resetting the counter. To avoid mistakenly passed barriers, the last threads exchanges the counter reference [194]. To implement the Lubachevsky barrier approach, two AICs are used [189]; on the SCC this, however, requires polling off-die AIC, therefore leading to contention. Hence, significant speedup is achieved by adding an exponential back-off to AIC polling during Signal phase. In

this thesis, it is leveraging the on-chip resources to implement efficient barrier synchronization, therefore, it did not use the AIC on the FPGA.

5.4.1 LUT Barrier Algorithm (LUTB)

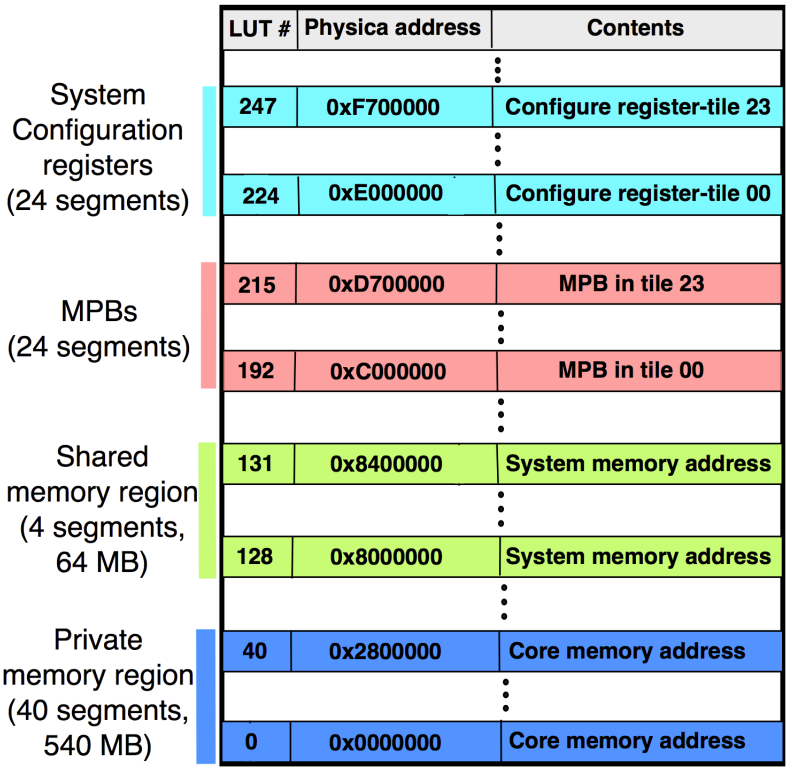


FIGURE 5.6: Lookup Table for 32GB memory system on the SCC

In the SCC, every core has a *lookup table* (LUT) with 256 entries as shown in Figure 5.6 used for physical-to-physical address mapping (32 bit core physical address to 64 bit system physical address) [4]. It is part of the configuration registers system that is mapped by a LUT entry also. The LUT is a shareable between all the cores and mostly used by operating system, but may be used also on application level. It is possible to change the contents of the LUT dynamically without causing problems to the core's memory management. The authors of [3] successfully implemented a copy operation by changing the mapping of LUT entries.

This raises the idea of using the LUT for barrier synchronisation. In order to achieve this in a sensible way it must be possible to propagate several signals onward to others without an increase in algorithm complexity. This is granted: not only can every core access its own or any other core's LUT, but also LUT entries are mapped by using `mmap()` in UC mode. As a consequence, by using LUTs it avoids the issues of ensuring

a consistent view of MPB. Moreover, the LUTs are located very close to the core and, as a result, it avoids the extra overhead of access off-die registers (e.g AIC). Therefore, LUTB is a new implementation that is using an one LUT entry in every core to track notify and release signals. This implementation therefore also avoids contention issues. It uses the same mechanisms for Entry and Signal phases as in the M-SSB algorithms: instead of using local memory for allocating the flags, it used a single LUT entry.

5.4.2 Chain LUT Polarity Barrier Algorithm (C-LUTB)

C-LUTB is a linear algorithm using chain mechanism for reducing the overhead of gathering the notify signals in the Entry phase of LUTB. To implement this algorithm, it needs to use one entry in each core. In the signal phase, the master thread waits to receive the notification signal from its higher neighbor and exchanges the LUT entry of the participant's last thread. Because of the LUT entry of the last thread is not used (no further neighbor in the chain), it uses this entry for releasing the slaves. All slaves wait for exchanging the state of the last thread's LUT entry that indicates a release signal.

5.4.3 Binary-Tree LUT Polarity Barrier Algorithm (BT-LUTB)

To reduce the overhead of gathering the notify signals in the Entry phase for C-LUTB, it exploits the binary tree scheme (such as in Section 5.3.1) in this algorithm.

5.5 Methodology and Micro-benchmarks

In Section 4.2.1, it avoids the traditional way to measure the overhead in barrier algorithms that is padded the time read before and after the barrier algorithms in the Master thread. It proposes to classify the overhead into two site, Master and Slaves to try covering all the time consumption in barrier schema.

Kumar et al reported on overhead of the barrier under three conditions: random load, impact of load imbalance, and effect on network of synchronization operation [195]. In addition, G.H. et al proposed to add a certain delay in critical section [196]. In this work, it therefore implemented four kinds of micro-benchmarks based on the above scenarios: *Pure Overhead*, *Impact of static load*, *Impact of random load*, *Impact of load imbalance*, and *Effect on NoC*. Moreover, it has reprogrammed the Pure Overhead micro-benchmark to analyze the *Impact of memory access mode* on the performance of barrier algorithms.

The *P*ure Overhead estimates the algorithm scalability with increasing synchronization traffic only, as illustrated in Section 4.2.1.

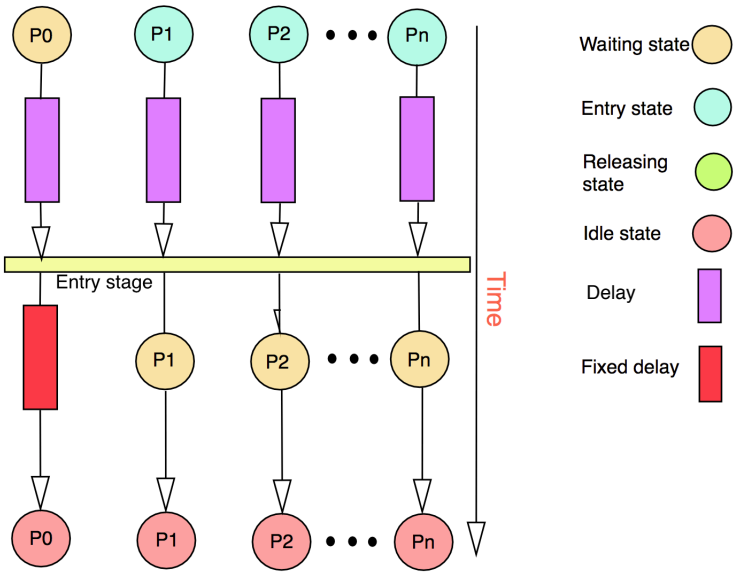


FIGURE 5.7: Impact of Delay implementation

Figure 5.7 depicts the implementations of three micro-benchmarks depending on added delay: it added delay in the master thread after the Signal phases to avoid interference with the barrier's next iteration. Here, it implemented versions for static and random loads similarly to the pure overhead micro-benchmark by adding static/random delay before the critical section of the slave threads. This staggers their arrival at the barrier point and inside the critical section in the Master thread's release phase. The random version basically approximates slight variations in the Exit phase. The static case is set up in a way that a core arrives with maximum efficiency in the Entry phase. Thus, these micro-benchmark measure the overhead of the slaves' arrival at the barrier as well as overheads of barrier synchronization due to loop scheduling. The average overhead time of the impact of load (static/random) are computed based on equations 5.1, 5.2, and the following:

$$Avg_{MD} = \frac{TotalDelayTime}{(No.ofIterations - No.ofIgnores)} \tag{5.1}$$

$$Avg_{SD} = \sum_{i \neq Master}^{No.ofthreads} \frac{TotalDelayTime}{(No.ofIterations - No.ofIgnores)} \tag{5.2}$$

$$O_L(CPUcycles) = \begin{cases} Avg_{MO} - Avg_{MD} & \text{if } thread \text{ is Master} \\ \frac{Avg_{SO} - Avg_{SD}}{(No.ofthreads)-1} & \text{else} \end{cases} \tag{5.3}$$

Where:

- $TotalDelayTime$: is the time of the barrier phases without any Static/Random load overhead.
- Avg_{MD}: is the average time of Delay (static/random) in the Master thread before entering the barrier.
- Avg_{SD}: is the average time of Delay (static/random) for Slave threads before entering the barrier.
- $O_L(CPUcycles)$: is the average load impact for Master or Slaves in CPU cycles.

Parallel programming experiments demonstrated that threads typically fail to arrive at the synchronization point at the same time for several reasons. Firstly, the workload may be unevenly distributed among the threads. Thus, certain threads consistently arrive late at the synchronization point, hence idling other threads and resulting in *load imbalance*. Similarly in OpenMP, all threads that entered the barrier might have to wait for a member of their team to carry out the work of a *single* construct. To determine the effect of load imbalance, all cores use the same delay except one core using a larger delay (2x Load). It chose the minimum delay size that is used by all threads to ensure that there is no interference between the two barriers. By using this method, it can measure the overhead of the last thread arriving; all the threads with smaller delay finish the first phase of the barrier and are waiting for the last thread using the larger delay. Therefore, to avoid unexpected barrier cost of the barrier, it is important to change the ownership for the larger delay in each iteration by using a round-robin approach (100,000 iterations per round). Under those circumstances, the larger delay has a measurable impact on the performance of micro-benchmark, which can be formulated as follows:

$$Avg_{MD_S} = \frac{TotalDelayTime_{Smaller}}{(No.ofIterations - No.ofIgnores)} \tag{5.4}$$

$$Avg_{SD_S} = \sum_{i \neq Master, \forall i \in smallerdelay}^{No.ofthreads} \frac{TotalDelayTime_{Smaller}}{(No.ofIterations - No.ofIgnores)} \tag{5.5}$$

$$Avg_{SD_L} = \frac{TotalDelayTime_{Larger}}{(No.ofIterations - No.ofIgnores)} \tag{5.6}$$

$$O_{IL}(CPUcycles) = \begin{cases} Avg_{MO} - Avg_{MD_S} & \text{if } thread \text{ is Master} \\ \frac{Avg_{SO} - (Avg_{SD_S} + Avg_{SD_L})}{(No.ofthreads) - 1} & \text{else} \end{cases} \quad (5.7)$$

Where:

- $TotalDelayTime_{Smaller}$: is the time of the barrier phases without any load overhead for threads used small load.
- $TotalDelayTime_{Large}$: is the time of the barrier phases without any load overhead for threads used large load.
- Avg_{MD_S}: is the average (small) Delay in the Master thread.
- Avg_{SD_S}: is the average (small) Delay for the Slave threads.
- Avg_{SD_L}: is the average (large) Delay in the long-delayed Slave thread.
- $O_{IL}(CPUcycles)$: is the average load balance impact for the Master or Slaves in CPU cycles.

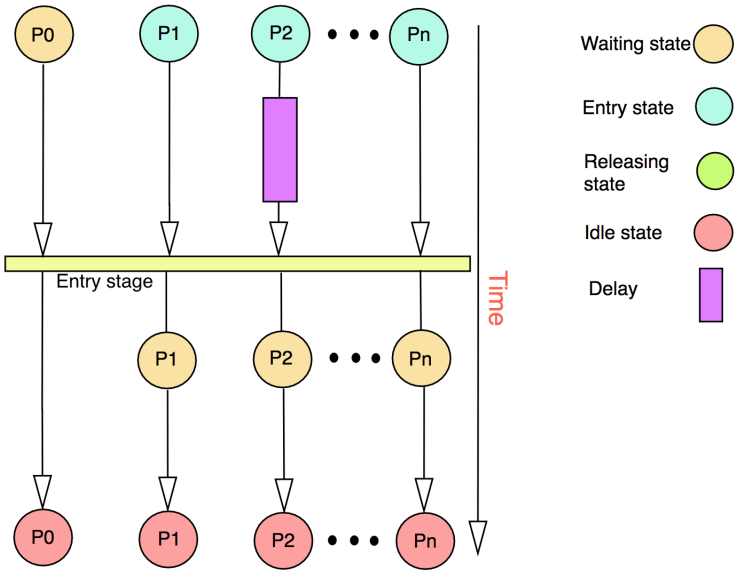


FIGURE 5.8: Effect of NoC implementation

Finally, it evaluates the effect of NoC traffic generated by the barrier operations for cores not participating in the barrier. All cores issue repeatedly the barrier code without any computation and communication except one core (the last numbered thread) which

does not use the barrier, performing read operations to an address in memory. Memory access can be done in two ways: access to an address allocated in DRAM (off-chip) and the access to an address allocated in SRAM (On-chip, e.g, MPB). The UC mode is used to access DRAM memory, while the MPBT mode is used to access SRAM. The last-numbered core sends a signal to the Master thread to release the other slaves. Thus, it can estimate the impact of NoC traffic that is generated by the barrier synchronization algorithms as depicted in Figure 5.8. It reported the time as shown in below:

$$O_{NoC}(CPUcycles) = \frac{T_{read} - T_{read\_out}}{(No.ofIterations - No.ofIgnores)} \tag{5.8}$$

Where:

- T_{read}: is the total execution time of read accesses in the last numbered thread inside the barrier phases.
- T_{read_out}: is the total execution time of read access in the last numbered thread without barrier effects.
- $O_{NoC}(CPUcycles)$: is the average overhead of NoC in CPU cycles.

Correspondingly, it shows an effect of memory mode accesses (UC and MPBT mode) that is used for implementing the barrier algorithms, which are depending on local memory usage on Pure Overhead micro-benchmark. In addition, the analysis includes the effect of varying the NoC topologies as the number of cores increases.

However, it added further criteria for measuring the algorithms' scaling efficiency. In this case, unity value means that no overhead (imbalance) occurs between Master and slave synchronization efforts.

$$E = \frac{Average\_Overhead_{(Master)}}{Average\_Overhead_{(Slaves)}} \tag{5.9}$$

Where:

- $Average\_Overhead_{(Master)}$: is the average overhead of an algorithm on the Master thread for all micro-benchmarks.
- $Average\_Overhead_{(Slaves)}$: is the average overhead of an algorithm on all slaves for all micro-benchmarks.

5.6 Performance Evaluation

The experimental results of the algorithms in Table 5.1 were generated using the default SCC settings which are depicted in Section 3.9. Where, it executed each barrier algorithm 100,000 times, determining the mean execution time with time measurement (CPU cycles) being only performed (and listed for) on the Master core.

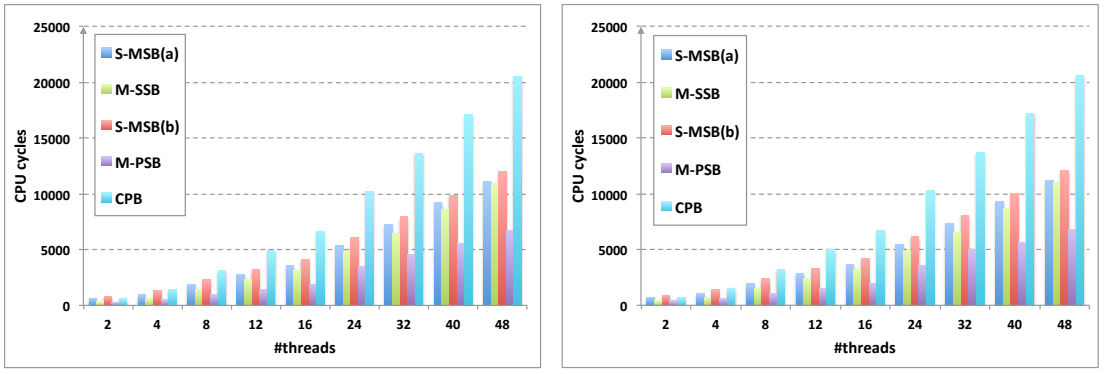
All micro-benchmark results in this section can be mainly divided into two groups which are Master Overhead (MO) and Average Slave Overhead (ASO) as illustrated in Section 4.2.1. The influence of the NoC is regarded separately. Therefore, the direct comparison of the barrier costs on Master and Slave side are determining the quality of the evaluated algorithms with regard to execution time and parallel speed-up.

TABLE 5.1: Barrier algorithms implemented

Algorithm	Description
S-MSB(a)	Linear barrier using MPB and local-get/remote-put approach
S-MSB(b)	Linear barrier using chain approach in Signal phase
M-SSB	Linear barrier using one array of flags are allocated onto their local MPB
M-PSB	Linear barrier using polarity exit approach and a single shared variable to terminate the slaves
CPB	Linear barrier allocating one array of flags onto their local MPB and using chain approach to gather and release them
BTPB	Tree barrier using polarity exit approach and a single shared variable to terminate the slaves and Binary tree to gathering them
D-BTPB	Tree barrier using Binary tree approach to gathering and releasing the slaves
LUTB	Barrier based on LUT to gathering the slaves and single shared variable to release them
C-LUTB	Barrier based on LU using a linear mechanism (chain) to gathering the slaves and a single shared LUT entry to release them
BT-LUTB	Barrier using Binary tree based on LUT in Entry phase and Binary tree based on flags are allocated in local MPB in Signal phase

5.6.1 Pure Overhead

It first wanted to determine the pure overhead of the barrier algorithm, resulting in the measurements show in Figure 5.9, Figure 5.10, and Figure 5.11. Figure 5.9(a) shows that the M-PSB's algorithm (Linear Barrier algorithm using polarity exit mechanism to gather entry signal) on the Master thread is approximately 53.4% and 39.5% faster than S-MSB(a) across 2 and 48 threads respectively. Under different number of cores, M-PSB achieves good performance. The Master thread overhead includes both, time for waiting for the slaves and time for releasing them on the SCC. The common point between these two algorithms is the gather phase as shown inTable 5.1. The single array of flag that allocated in Master thread's local memory. They differ with respect to the mechanisms used for release the slaves. Here, M-PSB uses a single shared variable with polarity mechanism for releasing the slaves. This approach shows benefits with a large number of cores and also when the location of cores allocated is not in the same coordinate (y-axis) of the Master thread. This also reflects that there is an effect of Network topology and routing mechanisms on the algorithm performance. An increasing number of cores shows however less effect on the overhead of the Master thread for M-PSB. In addition, this approach reduces the memory usage for allocating the flags.



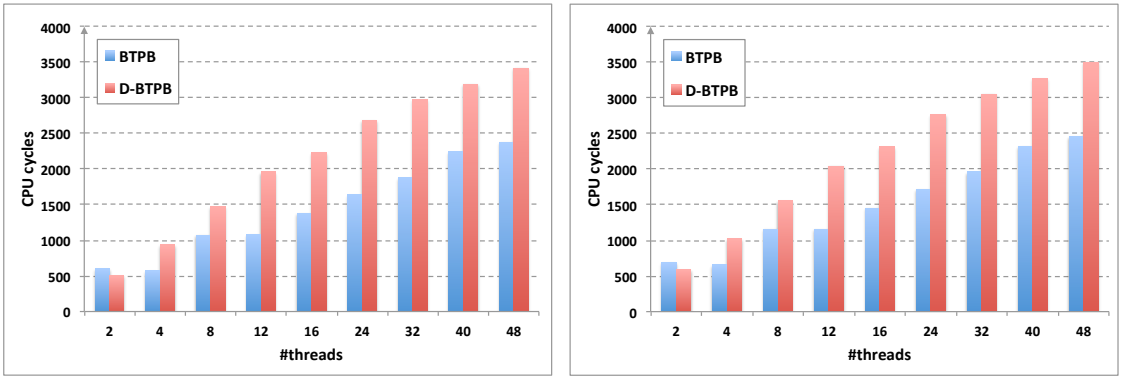
(a) Master Overhead

(b) Average Slaves Overhead

FIGURE 5.9: Pure Overhead of Linear algorithms

Figure 5.9(b) shows the average overhead for notifying the Master thread by the slaves and waiting time for the exit (release) signals. As it can see, the average overhead of the M-PSB approach (Linear barrier using a single variable allocated in Master's MPB to release slaves) is about 1.6 times lower than S-MSB(a). There is contention for notifying the Master because of notify flags allocated in single local memory (Master thread), and also no extra overhead for waiting the release signal come from Master thread. It can conclude that it is no jostle to poll the release signal on a single shared variable and also it doesn't need to reorder access to a shared variable. It shows no effects from the contention caused by accessing shared signal variables using M-PSB and the overhead

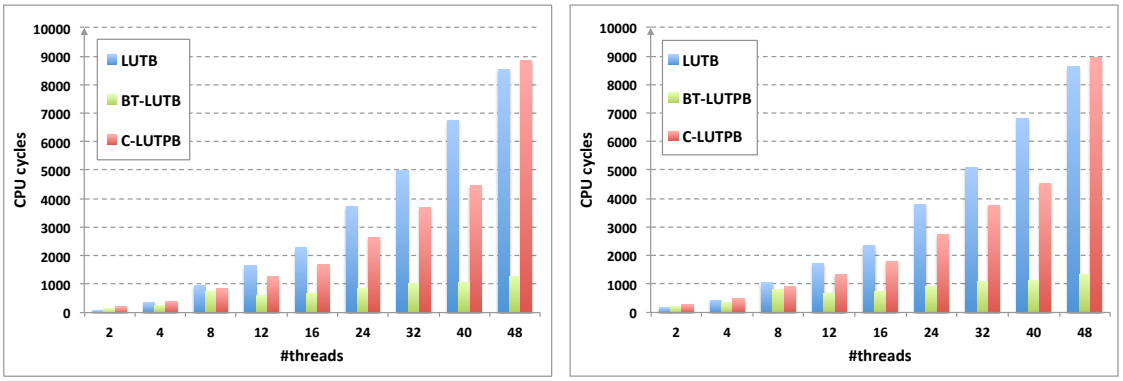
added by waiting for receiving the exit signal from the Master. To obtain an optimal barrier implementation based on linear algorithm and local memory utilize, it therefore can use approach of M-PSB. In the final analysis, the linear algorithms have added an negligible overhead in ASO more than MO.



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.10: Pure Overhead of Tree Algorithm

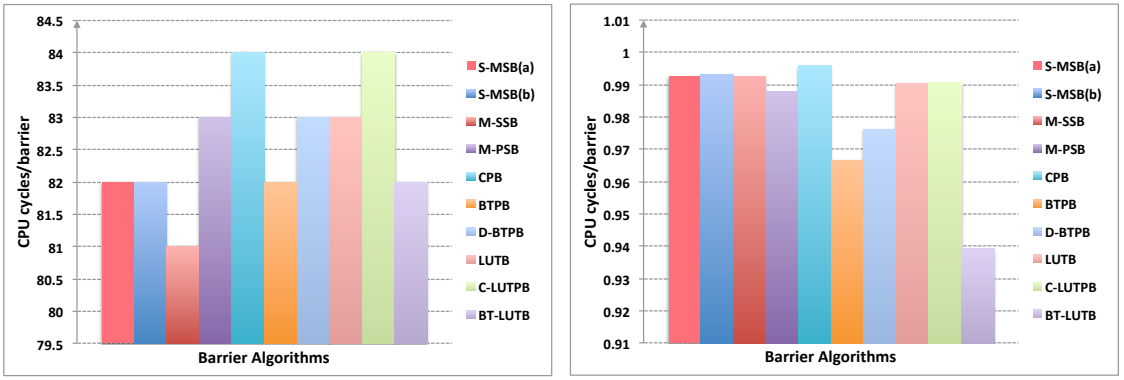
For the case of using tree algorithms, a stocky difference in the ASO between BTPB (tree barrier based on binary tree and polarity exit approach) and M-PSB algorithm occurs across 48 threads executed on at least 12 cores, as shown in Figure 5.10(b). Here, the BTPB approach achieved about 78% performance optimization compared to the baseline S-MSB(a). In the same fashion, the MO using BTPB (blue bar) shows major difference between M-PSB and M-SSB as illustrated by Figure 5.10(a), about 65% more than M-PSB on 48 threads. This indicates that neither using the Tree structure nor scheduling mechanism have an impact on the performance but may be instead add extra overhead to the algorithm as shown for the D-BTPB algorithm in Figure 5.10. This is a direct result of the x/y routing effect on resource-access behavior; the latter is mainly dependent on the y coordinate of a resource attached to the SCC's on-die network [189].



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.11: Pure Overhead of Barrier Algorithm based on Hardware Primitives

For the case of barriers synchronization based on hardware primitives, Figure 5.11 plots the observed overhead in the Master and Slave threads for varying numbers of cores. It clearly shows that BT-LUTB (based on LUT entries only) outperforms on all the other barrier algorithms (linear or tree). Using algorithms relying on LUT entries shows bigger benefits than linear and tree structures, as illustrated by Figure 5.11. For the core numbers >2, BT-LUTB shows significantly higher ASO than LUTB and LUTB. This results from the overhead added by aforementioned congestion and network effects.



(a) Master/Slaves difference

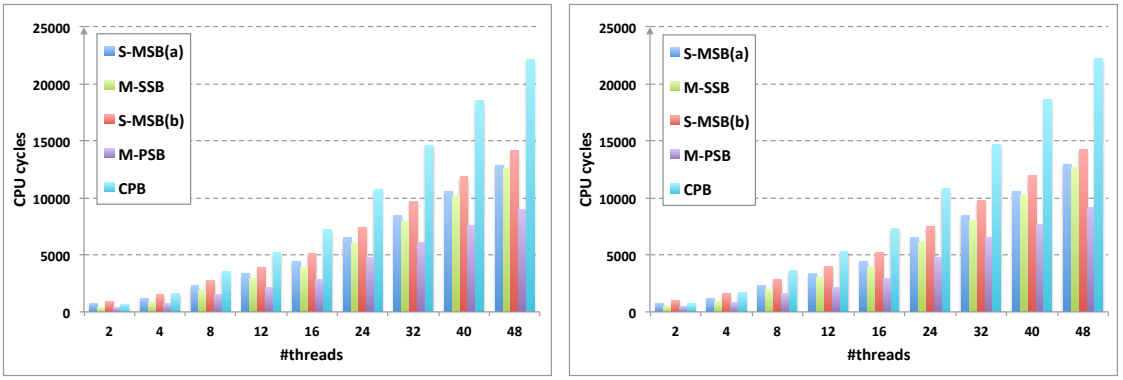
(b) Efficiency

FIGURE 5.12: Comparison of the Pure Overhead performance on 48 cores

Under those circumstances, optimizing the Master thread's Entry and Signal phases might improve the performance of the algorithm. Such optimization does not necessarily show likewise improvement on the Slaves' side, sometimes even the opposite effect might occur. Figure 5.12(a) shows the overhead difference ("efficiency") between the Master thread and its slaves for every algorithm. M-SSB shows the least difference in overhead, as compared to all the other algorithms. Therefore, the efficiency of this algorithms is high (close to unity value) as shown in Figure 5.12(b). As a consequence, this indicates that there is a kind of balance between the barrier algorithms' time consumption on Master and Slaves and a slight contention added by the algorithms.

5.6.2 Impact of Static load

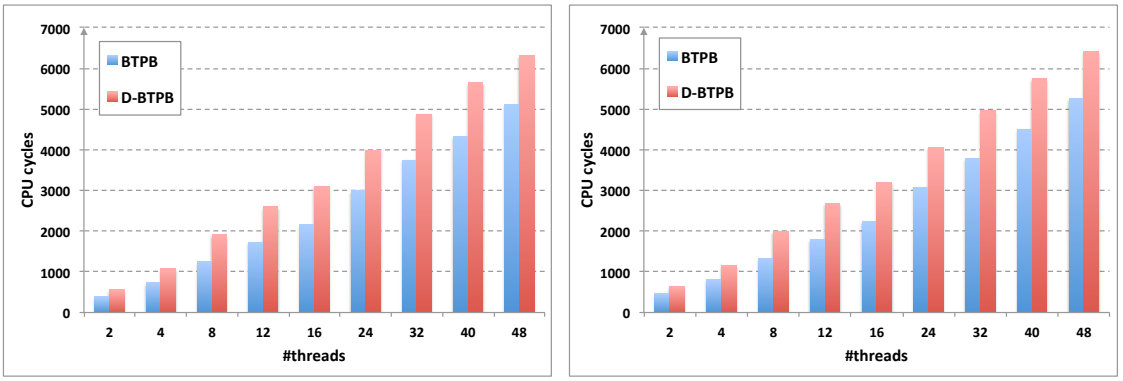
The results for each Barrier algorithm implementation are plotted in Figure 5.13, Figure 5.14, and Figure 5.15. Each figure depicts two curves corresponding to the implementations running on a different number of cores. The observation in Figure 5.13(a) is: linear implementations like algorithms M-PSB and M-SSB didn't benefit from adding fixed delay to the barrier and show extra overhead. There is a slight overhead difference between M-SSB and S-MSB(a); it also shows that no big impact on the overhead occurs for increasing the number of cores. Evidently, adding fixed delay to the barrier improves



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.13: Static load Overhead of Linear algorithms

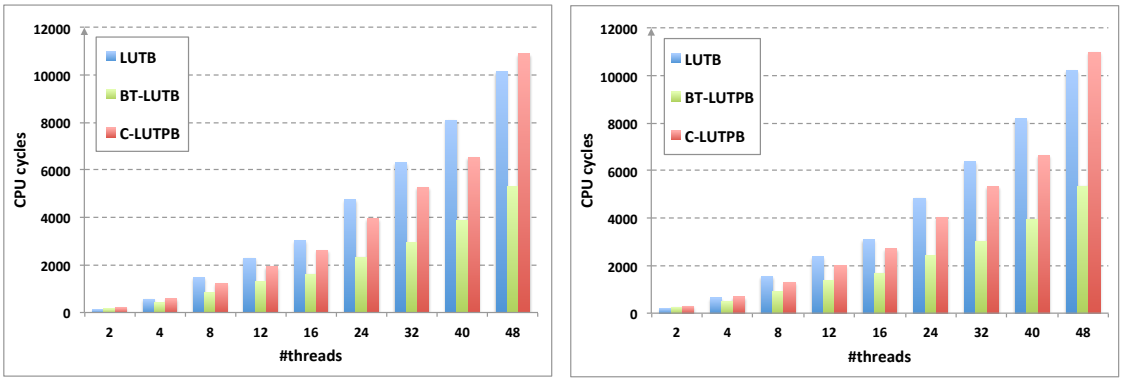
the performance of S-MSB(a) as compared with Pure Overhead results, and also for other algorithms. Comparatively, Figure 5.13(b) shows the behavior of S-MSB(a)/(b), M-PSB, M-SSB, and CPB. Here, M-PSB performs better because of less time required by the slaves for updating their poll flags as these are allocated in their local memories (MPB).



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.14: Static load Overhead of Tree algorithms

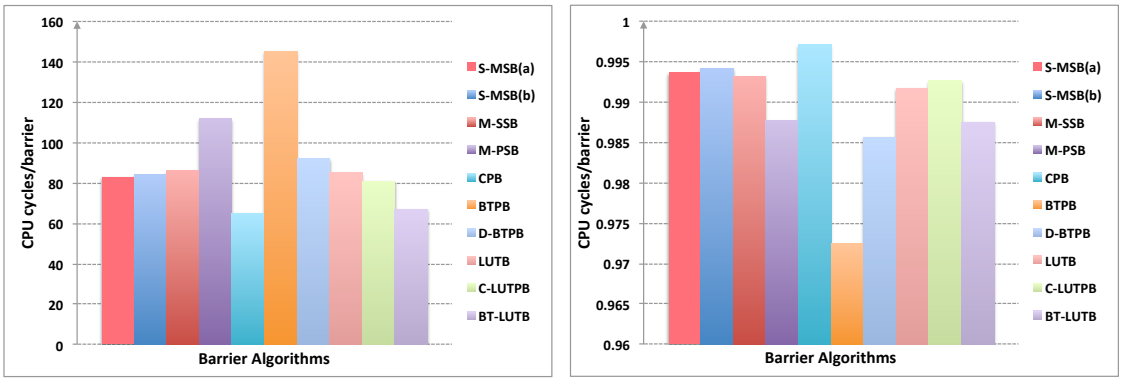
The effect of adding a fixed-length delay is evident also on the MO of Tree algorithms as shown in Figure 5.14(a). Here, it notes the impact of increasing core numbers which *increase* the overhead – which is the opposite of what was observed in some linear algorithms. But still the tree mechanism do perform better than linear schema. This results from the logarithmic scheduling and the resulting delivery of the notification signal. Consequently, this algorithm shows good performance with increasing core numbers, even above 48 cores compared with Linear algorithms. The ASO, as explained in Figure 5.14(b) shows similar effect of added delay with regard to the overhead.



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.15: Static load Overhead of Barrier Algorithm based on Hardware Primitives

The surprise was in Figure 5.15(a): the BT-LUTB algorithm's MO was affected by the increase in core numbers and add delay, it showed no improvement in performance. Unfortunately, this behavior was reflected the same for ASO where instead it notes an increase in the BT-LUTB overhead; the ASO of BTPB is slightly higher compared with ASO in the tree implementation case.

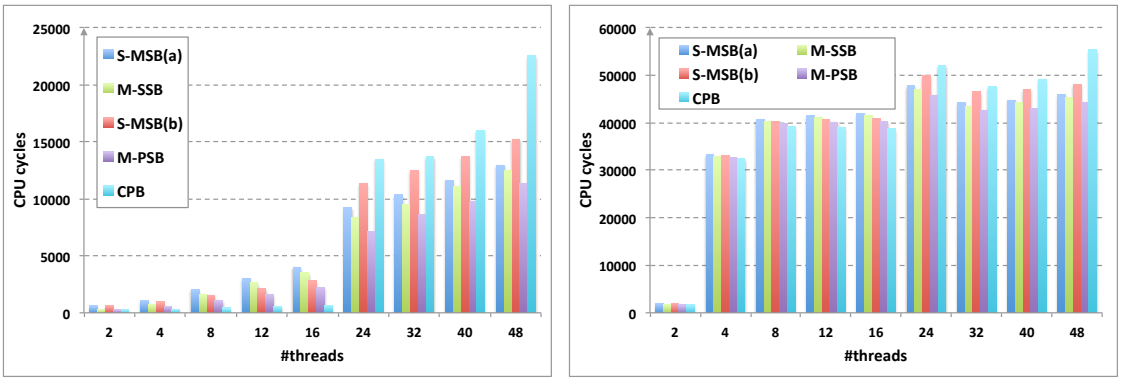


(a) Master/Slaves difference (b) Efficiency

FIGURE 5.16: Comparison of the Static Load overhead difference between master and slaves on 48 cores

As can be seen in Figure 5.16(a), the overhead difference between the Master thread and its slaves is not reduced for certain algorithms (M-PSB, M-SSB, BTPB, D-BTPB, LUTB, and BT-S-MSB(a)/(b)). CPB and BT-LUTB show an overhead reduction of approximately 23% approximately compared to all other algorithms. Accordingly, the efficiency of this algorithms remains high as shown in Figure 5.16(b), indicating that compared to the others this algorithm features a higher overhead balance between master and slaves than other algorithms. Also, the fixed-length delay is contributing to the increase efficiency of CPB and BT-LUTB algorithms as compared to Pure Overhead results.

5.6.3 Impact of Random load



(a) Master Overhead

(b) Average Slaves Overhead

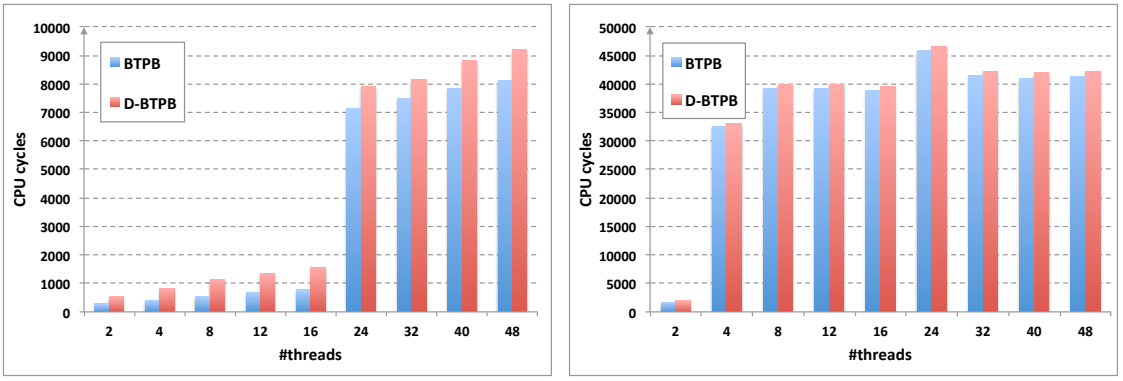
FIGURE 5.17: Random load overhead of Linear algorithms

Figure 5.17, Figure 5.18, and Figure 5.19 show the overhead when synchronizing a variable-length delay in a parallel block. The experimental results depict that the behavior of almost algorithms is a similar to its behavior when synchronizing a parallel work in fixed length. One interesting feature is that the linear barrier (CPB) shows better performance than the Linear, Tree, and hardware-based approaches, when number of cores <= 16. The CPB implementations featuring the exit polarity mechanism and LUT entries only do a better job than those have linear or logarithmic release approach. In addition, all the algorithms that implemented based on chain schema have the similar behavior of CPB implementation. For the case of core numbers >= 24, Master site has a clear impact and large on the performance of all algorithms. To relax the contention and the overhead, it believes that by applying a two master-like approach such as in [147] to those algorithms. Therefore, those algorithms depend on scheduling techniques such as tree implementation have less overhead consumption than other approaches. Similarly, ASO behaves the same way when core numbers > 2, otherwise, less number of cores and near the master thread have low overhead.

The comparison on the overhead difference for all algorithms are summarized in Figure 5.20. It can be seen that the barrier algorithms based on Hardware primitives (excluding the BTPB) showed substantially lesser overhead than the Linear and Tree barriers. As depicted in Figure 5.20(b), the CPB algorithm has higher efficiency as expected and as well it has less difference between the master and average slaves overhead.

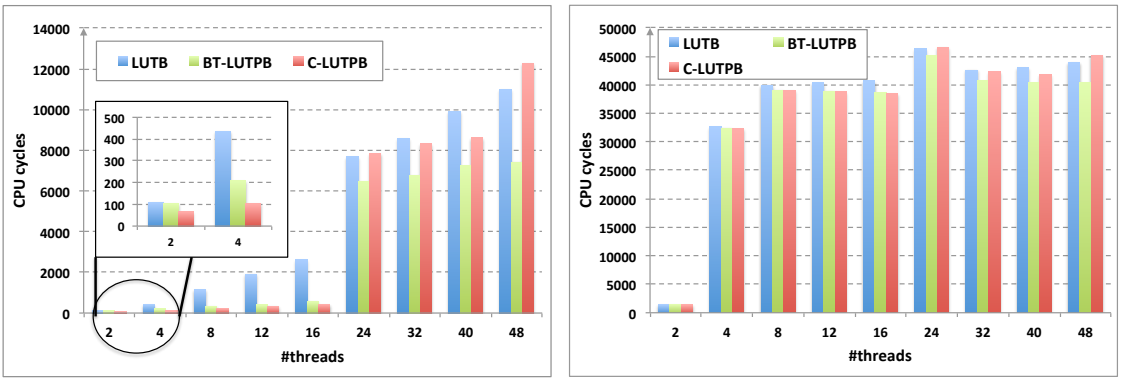
5.6.4 Impact of Load Imbalance

The overhead of barrier synchronization when all threads escape the barrier after the last thread arrived is shown in Figure 5.21, Figure 5.22, and Figure 5.23. Here, it used



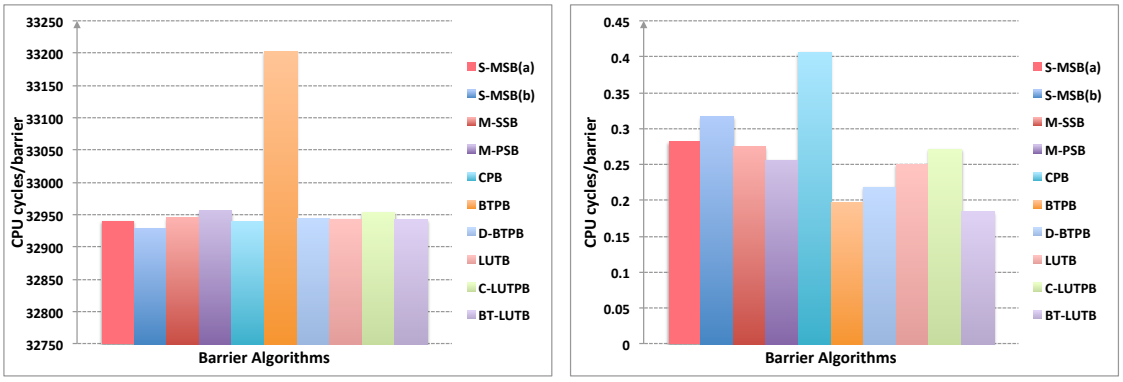
(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.18: Random load overhead of Tree algorithms



(a) Master Overhead (b) Average Slaves Overhead

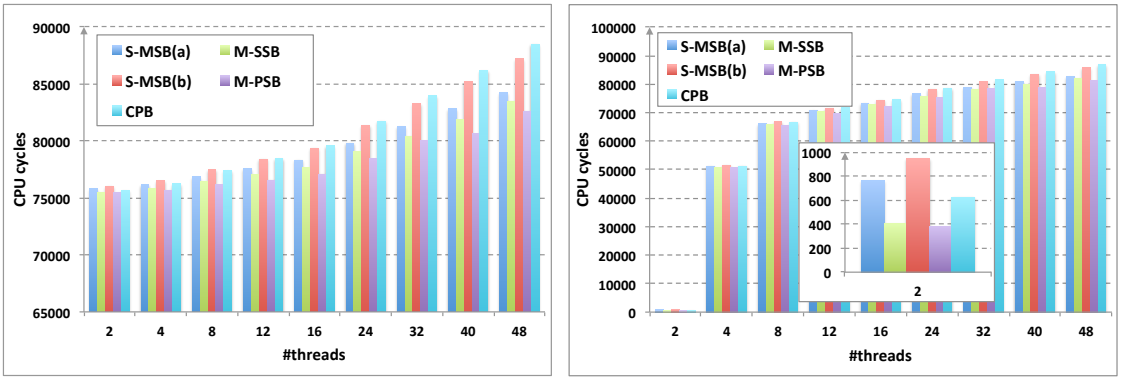
FIGURE 5.19: Random load overhead of Barrier Algorithm based on Hardware Primitives



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.20: Comparison of the Random load Overhead difference between Master and Slaves on 48 cores

the micro-benchmark explained in Section 5.5, now determining the barrier synchronization overhead as a function of both the particular barrier structure used and the load



(a) Master Overhead

(b) Average Slaves Overhead

FIGURE 5.21: Load overhead imbalance of Linear algorithms

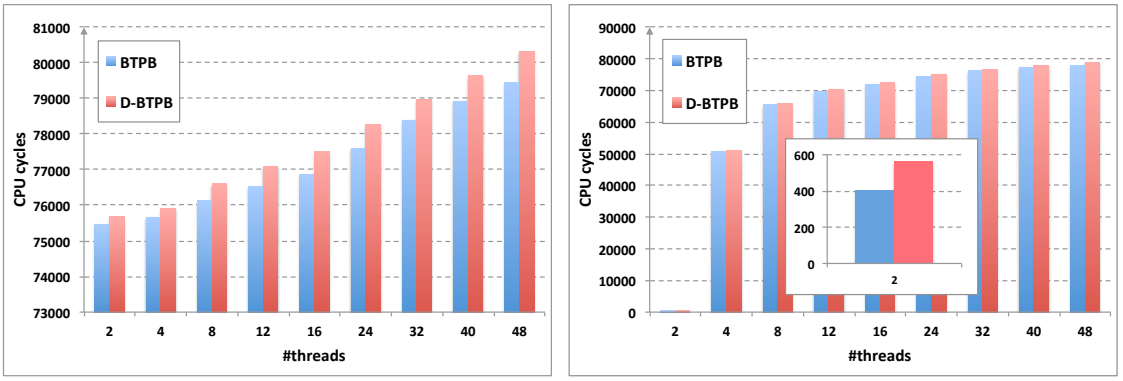
imbalance. It does so in order to demonstrate that synchronization mechanism and the load imbalance are not separated issues that can be solved independently. Instead, a maximum contention is added when developing a barrier mechanism by assuming the case of zero load imbalance.

Figure 5.21(a), Figure 5.22(a), and Figure 5.23(a) show that BTPB's MO benefits substantially from the load imbalance while others show only modest gains resulting in similar performance for all the barrier implementations. This has been reflected positively also on the ASO (Figure 5.21(b), Figure 5.22(b), and Figure 5.23(a)) as compared to the overhead determined in the previous experiments, although some of those algorithms exposed only few overhead. This leads us to the assumption BTPB may perform on par with the other barrier implementations in the case of applications showing certain load imbalance. Furthermore, there is a negligible overhead between BTPB and BT-LUTPB for 48 threads.

It did not evaluate efficiency of all algorithms in this experiment, as the efficiency is only determined for the case where all threads arrive at the barrier point simultaneously.

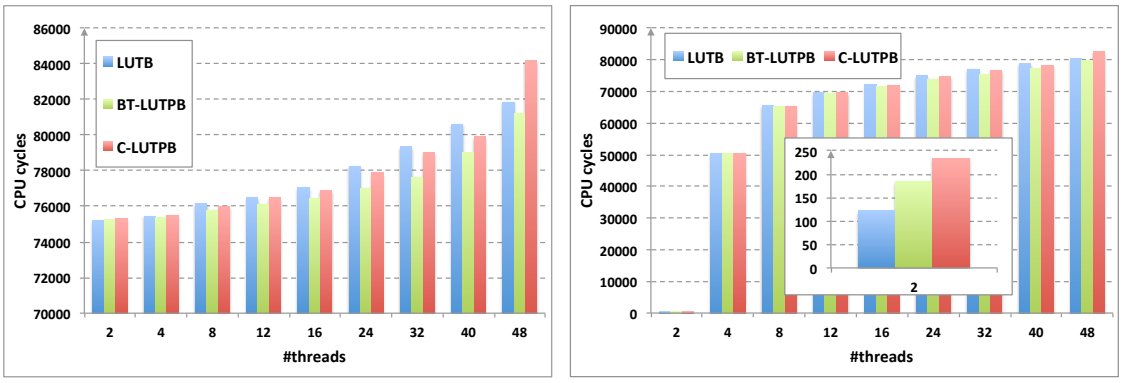
5.6.5 Impact of NoC traffic

The impact of NoC traffic for all algorithms are illustrated in the Figure 5.24. In this section, it allocated data into two address space; DRAM (off-chip) and SRAM (Master's local memory). The main goal of using the synthetic micro-benchmark with two different address portions is to monitor and analyze the barrier-caused network traffic, therefore also taking NoC traffic and according run-time into account. Based on the condition mentioned in the Section 5.5, just one core has access to the memory (on/off-chip) while others are waiting for a signal from this core. This ensures fair analysis of NoC effects



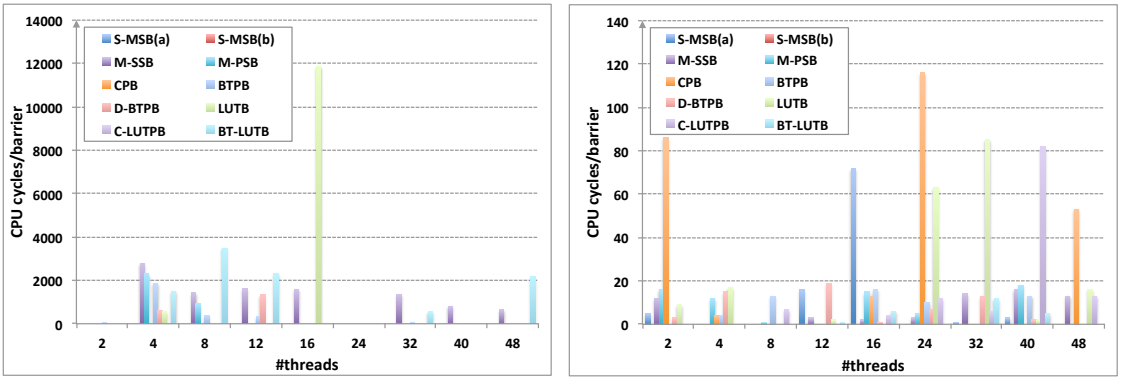
(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.22: Load overhead imbalance of Tree algorithms



(a) Master Overhead (b) Average Slaves Overhead

FIGURE 5.23: Load overhead imbalance of Barrier algorithms based on Hardware Primitives



(a) DRAM read access overhead (b) SRAM read access overhead

FIGURE 5.24: Impact of NoC traffic for Barrier algorithms

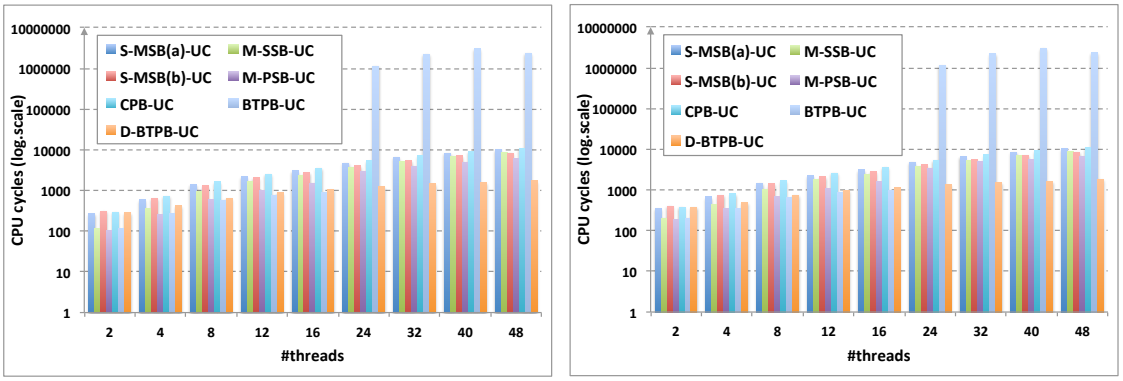
under the impact of barrier synchronization packets. In addition, the analysis includes the effect of varying the NoC topologies as the number of cores increases. Implicitly, this approach also takes also the memory controller into account (off-chip SDRAM access).

In Figure 5.24, what one sees here is that certain approaches to show *zero* overhead, meaning that no extra network traffic was generated by this algorithm and, consequently, no performance impact on single-thread reads occurs – regardless of increasing core numbers.

In fact, there is a negligible overhead occurs for almost algorithms, when participant's cores close to SRAM on-chip or memory controller. It still believes these algorithms have not been able to cause network contention; therefore, the dominant effect on the execution time is the memory access frequency in the figure.

Overall, BT-LUTB and algorithms based on the Polarity mechanism perform best among all the barrier implementations on the micro-benchmarks.

5.6.6 Impact of Memory Access Mode



(a) Master Overhead

(b) Average Slaves Overhead

FIGURE 5.25: Impact of UC-mode of Barrier algorithms

To study the impact of memory access mode, it is re-implemented the algorithms (Linear and Tree) that have access to local memory (MPB) for Pure Overhead micro-benchmark. Figure 5.25 shows results from this experiment. In this experiment, the UC mode reduces the overhead on MPB-based algorithms, also it significantly improves the M-PSB algorithm as shown in Figure 5.25(a). The D-BTPB-UC approach shows worse results by more than 49% compared to its overhead in the previous experiments. Figure 5.25(b) shows that CPB-UC reduces the overhead by 50% approximately for 48 threads.

The BTPB is prone to significant delays when the number of cores > 16 because of memory contention to access Master's MPB, as demonstrated in Figure 5.25. BTPB has a single flag allocated in Master's MPB to release all slaves which are concurrently accessing this flag and causing the congestion. While this effect is due to the uncached

accesses to the SRAM and specific to the SCC, the trend is observed in many many-core applications.

Namely, implementing UC mode contributes on reducing the overhead, because of the setting does not require to invalidate L1 cache lines before accessing the UC-mapped memory and flushing the WCB after write operations. In addition, there is a slight overhead difference between M-PSB-UC and D-BTPB-UC; one also notes that there is less impact on the overhead with the increasing number of cores.

5.7 Summary

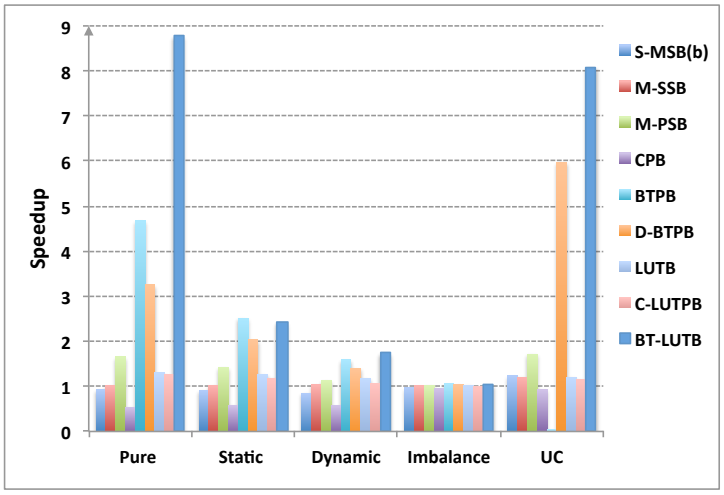


FIGURE 5.26: Speedup of several micro-benchmarks performance against the baseline for 48 cores

In this chapter, my goal is to analyse the different barrier algorithms' performance (as explained in Table 5.1) with respect to varying NoC topology sizes (as the number of cores increases), impact of workload (static, dynamic, and imbalance) for different topology sizes, and the complexity trade-off of the barriers combined with different NoC topologies and hardware primitives (MPB, and LUT) with different memory mode access. All barrier algorithms presented here directed towards a many-core on-chip. The barrier requires both shared places which can be accessed by all cores and private access with any conjugation. Therefore, it implemented several algorithms which are graded in complexity and NoC topology in this work.

In the process of designing the algorithms, three basic concepts used. First, some algorithms require only boolean variables in shared memory in array to avoid using spinlock routines that provide atomic read/write access to shared memory locations. Second, the communication pattern of the gather or release phase may be either linearly or tree structured. Finally, barriers may have symmetric gather and release phases, or

the release phase may be implemented as broadcast identifying a reversal of polarity. Consequentially, the hardware requirements for designing these algorithms are quite minimal. Only the availability of shared memory or register is required by half of the algorithms. In addition, using polarity-based mechanisms in release phase will reduce the overhead of reinitialize the barrier so that will function properly on its next iteration. Namely, the barrier will move to next iteration with one polarity state and the master core doesn't need to know the last core got the release signal or not.

However, achieving efficient barriers with low overhead into existing parallel programming models tends to result in only minor improvement in the speedup if these programming models were one baseline barrier to begin with. Reducing the barrier's overhead time delivers instead a different payoff: the size of work loads that may profitably be parallelized is decreased.

$$Speedup = \frac{Avg_{MO}(S - MSB(a))}{Avg_{MO}(*)} \tag{5.10}$$

Equation 5.10 shows the formula for speedup when considering only MO of S-MSB(a) barrier, where $Avg_{MO}(*)$ is the MO for $(*)$ barrier algorithms to execute micro-benchmarks based on different scenario of load distribution. Figure 5.26 shows the speedup of all barrier algorithms against S-MSB(a) for 48 cores only.

When synchronizing 48 cores with adding workload (static/dynamic), the effective execution time of BT-LUTB and BTPB on machines structured in two-dimensions is shown. Obviously, adding workload (static/dynamic) to some algorithms has contributed to increasing the overhead as explained in Section 5.6. Of course it will also be reflected on the overhead of slave threads. The experiment with dynamic load shows extra overhead added than fixed length work. The variance in these times is mostly due to linear and tree algorithms, whose performance is quite dependent on the type of work that they synchronize. The tree-based polarity barriers had much more stable execution times across the experiments. Here, the primary advantage of the tree approaches is their logarithmic depth. As the number of cores in mesh becomes large, this advantage becomes overwhelming, as demonstrated in Figure 5.26.

While, BT-LUTB reduces the overhead of communication in case of there is no work load adding between participated cores in parallel block. The load imbalance has a contribution to overhead reduction. For the case of tree algorithms, the resulting algorithms show higher complexity and also different topology behaviour compared to linear algorithms. The logarithmic depth resulting from the binary tree and different workload did achieve a desired performance improvement. It is clear in case of pure overhead experiment in UC mode, D-BTPB achieved higher improvement in the performance as

compared with BTPB. But, BT-LUTB is still achieving high optimization in overhead because of access to LUT entries removes the contention in NoC. Namely, register or local memory, adding to the tile or a core node has access apart from NoC, removes a major part from congestion to access the synchronization flags.

This work addressed the problems associated with using blocking synchronization algorithms based on synchronization flags. These problems include complexity, high overhead, topology size, locality, high contention, and workload effect. Optimistically synchronized configuration registers such as BT-LUTB allow programmers to avoid these problems. The topology size has a quite an influence on the BT-LUTB barrier, while the load imbalance has only small impact on the overhead; the latter, however, is still lower than for other algorithms. The performance overhead introduced by the tree topology as in BT-LUTB and BTPB is the highest for the considered configurations. This is because of the NoC routing protocol effect that corresponds to the tree approach.

Henceforth, the BT-LUTB is the best barrier synchronization which allows more than 88% (MO in Pure Overhead) faster synchronization than the baseline S-MSB(a) implementation and approximately 46.6% (MO in Pure Overhead) faster than the typically well-performing BTPB algorithm.

Chapter 6

The Relevance of Architectural Awareness for Efficient Fork/Join Design

Several recent manycores leverage a hierarchical design, where small-medium numbers of cores are grouped inside *clusters* and enjoy low-latency, high-bandwidth local communication through fast L1 scratchpad memories, as explained in Section 2.1. Several clusters can be interconnected through a NoC, which ensures system scalability, but introduces *NUMA* effects: the cost to access a specific memory location depends on the physical path that corresponding transactions traverse. These peculiarities of the Hardware must clearly be carefully taken into account when designing support for programming models.

In this chapter, it studies how architectural awareness is a key to supporting efficient and streamlined fork/join primitives [197]. Then, it compares hierarchical fork/join operations to the "flat" ones after optimizing its performance, where there is no notion of the hierarchical interconnection system.

6.1 Motivation

Although many-cores allow tremendous performance/watt improvements, increasing continuously the number of cores in the system has resulted in an increase in the complexity in software and hardware [1]. Effective programming abstractions are key to tackling the increased system complexity, aiming at delivering both ease of application development and effective usage of the system's huge available parallelism. This, of

course, requires to employ all processors for most of the time. Therefore, it is necessary to provide efficient coordinated execution of a multi-threaded application on the system cores.

One of the most widespread programming paradigms for shared memory systems is the *fork/join* execution model. A parallel program starts as a single thread of execution, then when parallelism is available a thread *team* is *forked*. At the end of the parallel computation threads *join* on a barrier, then the execution resumes on a single thread. The fork/join model is adopted by the popular OpenMP [34] as well, the *de-facto* standard for shared memory programming. Barrier synchronization (thread *join*) has been recognized as an important source of the performance degradation in the execution of parallel programs [175, 177, 179, 180, 198]. A significant fraction of algorithms is dedicated to efficiently reduce the overhead of barrier synchronization with OpenMP constructs as illustrated in Chapter 4.

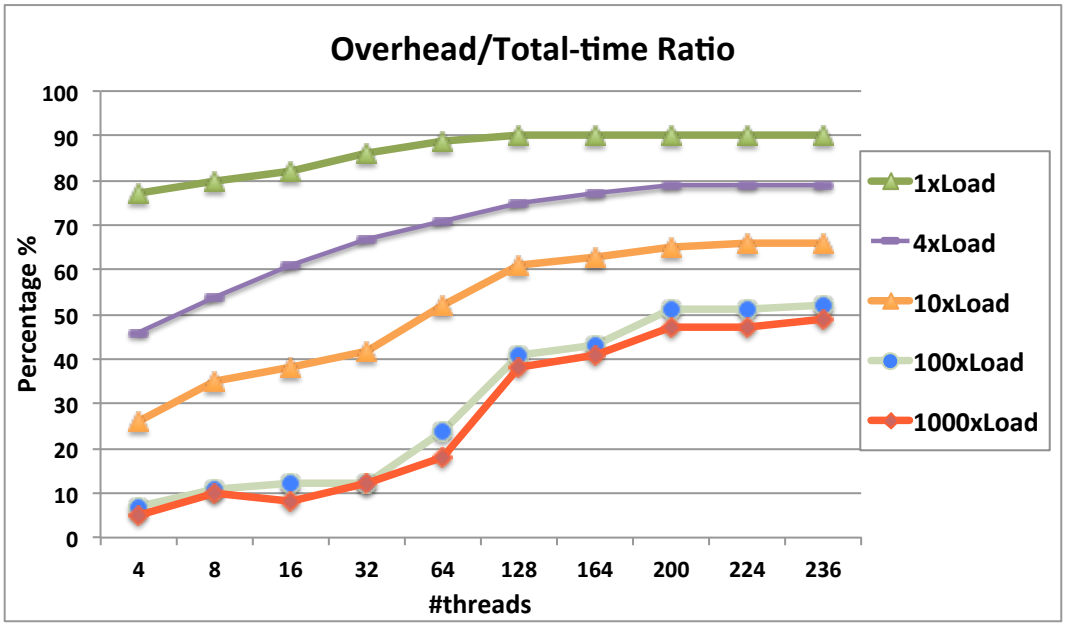


FIGURE 6.1: The fork/Join overhead in Xeon Phi [5]

Similarly a poorly scalable *fork* operation is bound to prevent effective program parallelization on a multi-core [182, 199, 200] and a many-core [201, 202] systems for OpenMP's fork/join. Figure 6.1 shows the percentage ratio of the overhead of fork/join model based on the total execution time versus the number of threads spawned in Xeon Phi (system overview explained in Section A). The EPCC benchmark [182] is compiled to run in "native-MIC" mode with different work load by using OpenMP* [172]. It found that the OpenMP overheads are typically higher when enabling fine-grain parallelism (green) in OpenMP programs and scale poorly for the large number of threads. Therefore, low overhead plays an important role when the amount of parallel work is small, because the overheads may quickly exceed the benefit of this limited form of parallelism.

Moreover, with increasing the load granularity (red), still the overhead of adding more threads to the application has significant impact on the performance. Namely, adopting *architecture-agnostic* algorithms to recruit a large number of threads during *fork*, and to synchronize them during *join* is subject to linearly increasing latencies, which quickly make these abstraction prohibitively costly for a many-core system.

In this chapter, a novel technique presented to explore the benefits of adopting a hierarchical approach to creating and synchronizing thread teams. The proposed fork/join algorithm assumes a many-core platform template organized as a set of clusters, and consider as main parameters the number of such clusters and the number of cores within each cluster. Based on this information thread recruitment (or synchronization) executing in consecutive steps. First, an outermost team is created, with as many threads as clusters. Then, each of these threads is involved in the creation of local thread teams within each cluster. Multiple inner teams are created in parallel over different clusters, thus reducing the overall *fork* (*join*) latency. The logical clustering considered in the hierarchical algorithm does not need to match the physical clustering in the platform. For many-core architectures where the number of physical clusters is very high, while only a few processors per cluster are present (i.e. SCC and Tilera), it may be more convenient to consider larger " virtual" clusters. Hierarchical fork/join is implemented on the Intel SCC, comparing it to architecture-agnostic fork/join (flat) and exploring different logical clustering schemes.

However, the rest of this chapter is organized as follows. Section 6.2 introduces the fork/join mechanism and the issues in its applicability to clustered many-cores, and the flat implementation with optimization techniques are described in Section 6.3. Section 6.4 describes the hierarchical fork/join scheme, and Section 6.5 provides the details of its implementation on the SCC platform. Section 6.6 presents the experimental evaluation. Then, it will discuss the related work in Section 6.7 and the summary in Section 6.8.

6.2 The Fork/Join Execution Model

Fork/join is a popular parallel execution model for shared-memory systems, implied in several higher level programming abstractions (e.g., OpenMP [34]). Figure 6.2(a) depicts the theoretical Fork/Join execution model. The program initially executes sequentially within a single thread, referred to as the *Master thread*, until it encounters a request for *forking* a parallel thread *team*. The additional worker threads are referred to as *slaves*. Blue and orange boxes in the figure highlight the sources of overhead on the master side to fork and join additional threads, respectively. Some overhead is also

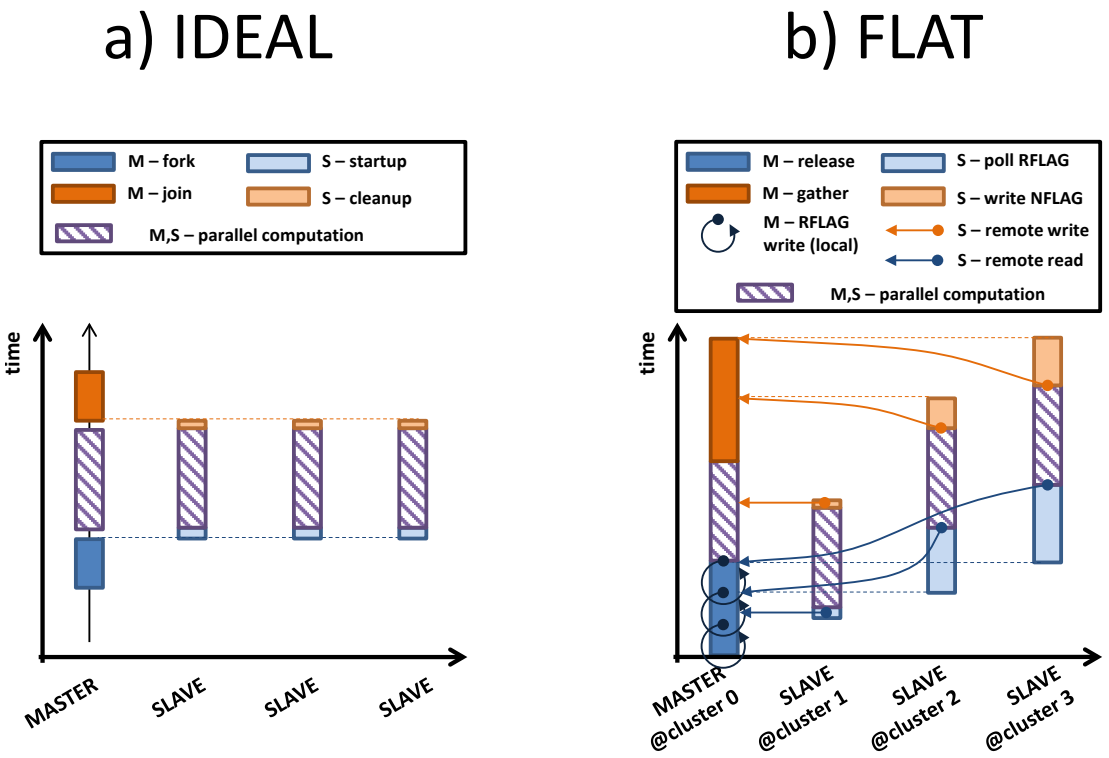


FIGURE 6.2: Fork/Join Model

present on the slave side (light-blue and -orange boxes), for parallel execution startup and cleanup. The striped boxes indicate the actual parallel work, ideally executed in a perfectly aligned and balanced manner among parallel threads. Intuitively, the smaller the blue and orange boxes, the better capability of the system to enable fine-grained parallelism.

There are two main approaches to practically implementing fork/join support. The first is based on dynamic thread creation (DTC), the second leverages a fixed thread pool (FTP). DTC is very flexible, but expensive both in terms of space (memory footprint) and time [166]. In a resource-constrained platform such as the targeted embedded manycores this approach may quickly run out of memory, and the resulting time overheads would disallow fine-grained parallelism. In addition, this approach would require dedicated abstraction layers to allow threads on different cores to communicate in cluster on-chip system such as SCC. FTP creates lightweight threads (typically as many as the number of processors) at system start-up, and "docks" them on a "pool" of idle workers. In this way a fork operation boils down to recruiting threads from the idle worker list, providing a function pointer and releasing them from the pool. Similarly, the join operation consists of gathering threads on the pool and cleaning up bookkeeping data structures.

My work is based on the FTP approach [45, 167, 201]. Here, a Master-Slave barrier construct is used to implement fork/join. At system startup the master *joins* all slaves on this barrier with a *gather* primitive. This is achieved by inspecting local flags to each slave, where they notify their arrival on the barrier (NFLAGS). The *fork* operation is implemented with a *release* primitive on the barrier. Every slave polls on a local release flag (RFLAGS), where the master signals that the corresponding thread has been "forked" (i.e., recruited into a parallel team). Figure 6.2 shows how the ideal fork/join model and associated sources of overhead change when this architecture-agnostic procedure is mapped on a cluster-based manycore. It is evident that there are three main sources of overhead when applying the simple mechanism on cluster-based many-cores:

1. Releasing slaves is done sequentially on the master, which lengthens the duration of the fork operation significantly for a large number of cores.

2. When the associated communication crosses the boundaries of a cluster, NUMA effects are present, which further lengthen the operation.

3. The actual start time of different threads in a parallel team may be significantly dis-aligned. For fine-grained parallel workloads this may have an effect on the overall parallel region duration, as the fork/join model imposes all threads to wait for the slowest thread before execution can proceed past the parallel region.

Similar problems are present also during *join*. The most naive approach to supporting fork/join on the target platform is that of implementing the basic execution model described above with no concern about NUMA effects. It calls this implementation "*flat*", as no notion of the hierarchical interconnection system is captured by the algorithm. To support flat fork/join, the algorithms in [167, 201] are extended to simply account for a larger number of processors. Flat fork/join will be used as a term of comparison to the architecture-aware schemes that it proposes in the following.

6.3 Flat Fork/Join Optimization, Why?

Consider the overhead cost for the fork/join parallel region implementation depicted in Figure 4.15. The cost of overhead is increased by having a large number of threads that included in parallel region, regardless of the way to distribute workload over threads, as shown in Figure 6.7. All these micro-benchmarks, depending on added delay such as explained in Section 4.2.2.

Here, it uses four kinds of micro-benchmarks to estimate the overhead that caused by fork/join mechanism: *Static Load*, *Imbalance Load*, *Dynamic Load*, and *Load Balanced*.

In *dynamic load*, the static load extended by distributing random load between threads. As a consequence, this schema allows to generate scene with a variable grain size of workload distributions. In this case some of threads will need longer time to complete their work than the rest, which delays the processing time of the overall system.

The core of OpenMP, work sharing directives such as *#pragma omp for* are responsible for distributing the workload between the threads and it's granularity depending on the number of threads. *Load balance* assumes a constant workload is divided equally between threads on parallel region similar to static schedule in loop parallelism. The workload in each thread has a size of $((problem\_size)/(number\_of\_threads))$. This gives a *block* decomposition, where each thread is held only a part of the problem to process. As a consequence, increasing number of participating threads in parallel region, resulting in an increase significantly in performance.

Before discussing the overhead cost of fork/join OpenMP model in Section 6.3.3 and why it needs the optimization, more details about the Flat implementation and the optimization techniques illustrated in Section 6.3.1 and Section 6.3.2 respectively.

6.3.1 Flat Implementation

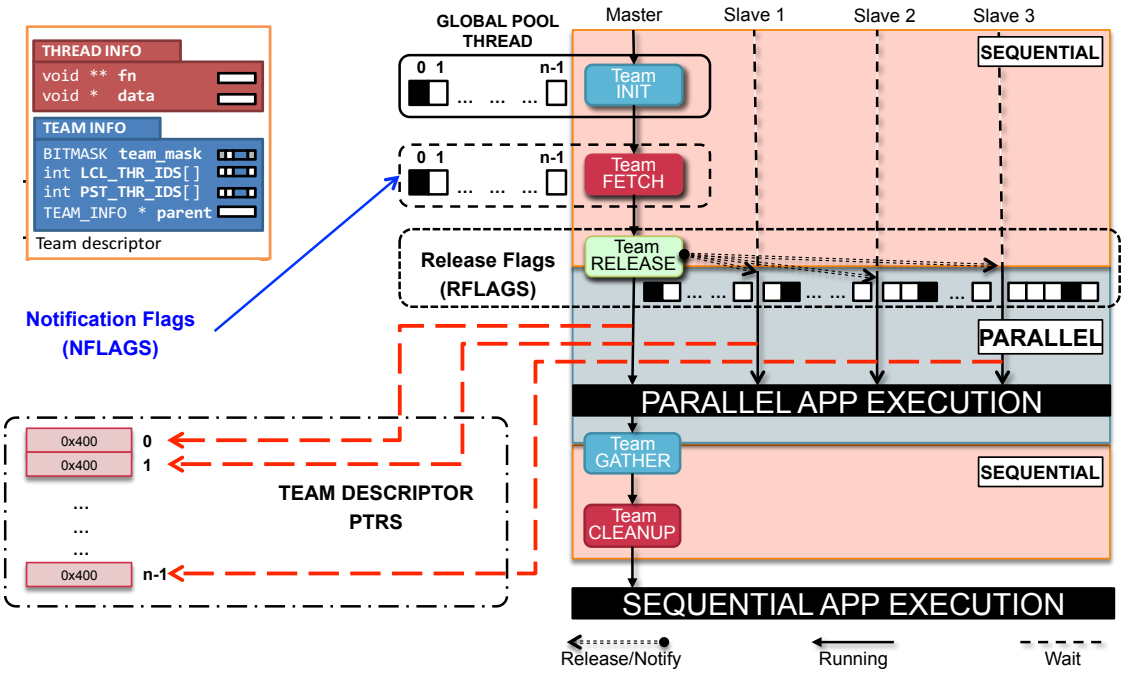


FIGURE 6.3: Thread landing, Synchronization, and Team descriptor

The implementation of OpenMP model is based on the FTP as depicted in Section 4.1.2, that relies on a custom micro-kernel code [105] executed by every core at

start-up. Master and slave threads execute different code based on their core IDs. After system initialization, the Master core jumps to execution of the parallel program's master thread, while the slaves wait for activation (fork). When the Master encounters a parallel region, it invokes the runtime system, points the slaves to the parallel function, and triggers execution. At the end of the parallel region, a global barrier synchronization step is performed. Here, Master core creates as many threads as cores with a private stack and a unique ID. It can call these threads *Persistent* because they will be re-assigned and used to parallel teams as needed, then will never be destroyed and non-preemptive as well.

To implement *flat* fork/join on the SCC platform, a *FTP* approach employed similar to those described in [167, 201], as shown in Figure 6.3. It allocates all the support data structures on the on-chip SDRAM memory local to the master thread: i.e., the master thread's MPB on the SCC. The team descriptor that contains the necessary information for parallel threads to execute (pointer to the parallel code and shared data) is allocated on the off-chip shared memory on the SCC.

Here, it promotes the Master is the lowest ID that will be running all the time. Of course, the Master is directly responsible for generating the topmost level of parallelism. While the rest of the threads (slaves) are landed in the global thread pool, waiting for the master tread to give them the work. Here, the slave executes a micro-kernel code where they first notify its readiness on its private location (based on its ID) of NFLAGS, then it waits for work to do on a private flag of another array (RFLAGS). However, to reduce the cost for master and slave polling activity when idle – namely read operations on the NFLAGS and RFLAGS data structures – it allocates each of them in the local on-chip memory (MPB). The status of each slave in the thread pool (idle/busy) is commented in a third global array, known as the *global pool thread*. When a master thread gets in a parallel region, it fetches threads from the pool and points them to a work descriptor. Besides fetching threads from the global pool thread, creating a new parallel team by allocating and initializing the information in a *team descriptor* (Team INIT in Figure 6.3). This descriptor contains two kinds of information:

- *Thread Information*: Two pointer variables are the code of the parallel function and its arguments.
- *Team Information*: This data structure maintains two local arrays. The *LCL_THR_IDS* array is used to index the persistent thread IDs and hold the corresponding local thread IDs. The *PST_THR_IDS* array involved the whole team information that used for services such as joining threads and updating the status of the pool descriptor. It also keeps the dual information: it is indexed with local thread IDs and returns a persistent thread ID.

After the master thread has filled all its fields in team descriptor, by storing its address in a *TEM_DESC_PTR* array, as shown in Figure 6.3 where one location per thread.

At the end of parallel region, the global barrier synchronization is used to gather all slaves (Team GATHER in Figure 6.3). After all slaves arrivals the barrier line, as the master thread clean all the fields in the data structure (elastic metadata) and gather all results of the parallel computation, the slave goes to waiting state for the next parallel region.

6.3.2 Optimization Techniques

This section explains in details the optimization techniques of Flat implementation for OpenMP fork/join model.

6.3.2.1 Synchronization Primitives

As explained in Section 6.3.1, the fork/join model imposes two synchronization events per parallel loop. Consequently, the costs of barrier for fork/join model can be high, especially when the application has nested loops such as parallel inner and sequential outer loops. The simplest way of enforcing synchronization in fork/join model is through the use of *barriers*. Namely, a barrier prevents all threads from leaving the barrier until all threads have reached that barrier. Henceforth, the synchronization of threads provides a useful way of preventing the *race condition*. A barrier is used to control and ensure all slave threads have completed before the master thread can continue. Control synchronization reduces parallelism in a program by forcing threads to wait until a certain condition holds.

Therefore, one needs to implement the barrier by the low cost of the overhead as an one important rule to produce an impressive programming model. As explained in Chapter 5, many techniques are proposed and validated to improve the performance of the barrier implementation. The first way of optimization mechanisms is the S-MSBO that represents optimization version of the original implementation of S-MSB primitives. It aims to show how much the barrier overhead hurts the performance for a several scenarios of application.

As shown in Figure 6.4 and Figure 6.5, the S-MSBO synchronization implementation reduced the overhead of the fork/join design. The influence of S-MSBO on the *Team GATHER()* primitive has grown exponentially with the number of cores used. In addition, the S-MSBO reduced the contention effect in NoC that is reflected negligibly as well

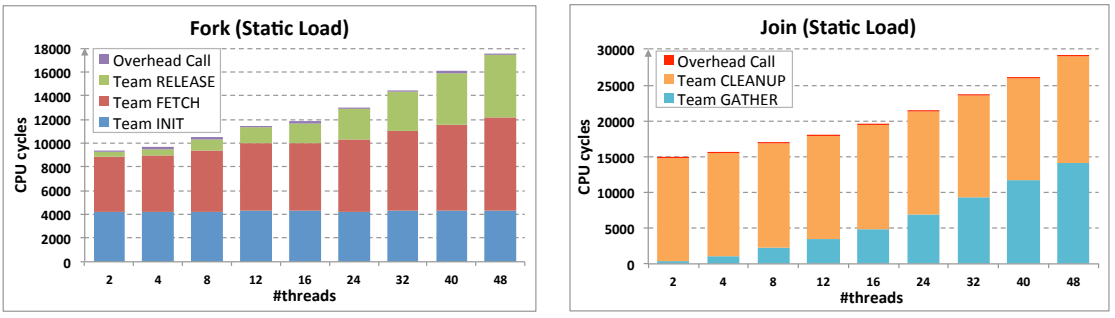


FIGURE 6.4: Cost of Flat fork/join model

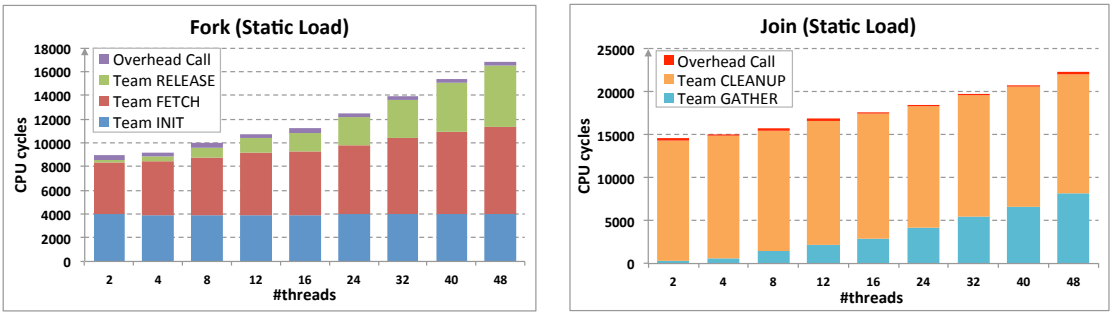


FIGURE 6.5: Cost of S-MSBO-based fork/join Optimization

in the overhead of *Team INIT/FETCH()*. Although, the overhead reduced ≈5000 cycles in join implementation, the total overhead for fork/join model is still high. Therefore, it needs to reduce other overhead that accompanied with INIT, FETCH, and CLEANUP of threads team.

6.3.2.2 Memory Allocation

In fact, many-core architectures have memory hierarchies and by exploiting the spatial and temporal locality can lead to better performance. Typically, many-core consist of a fixed size of local on-chip shared fast memory (such as MPB on SCC) and a large global off-chip shared memory that's accessible by all the cores. The shared local on-chip memory is faster to access than the global memory but it adding more congestion to the network. In such systems with explicitly managed memory performance is awkwarder to achieve.

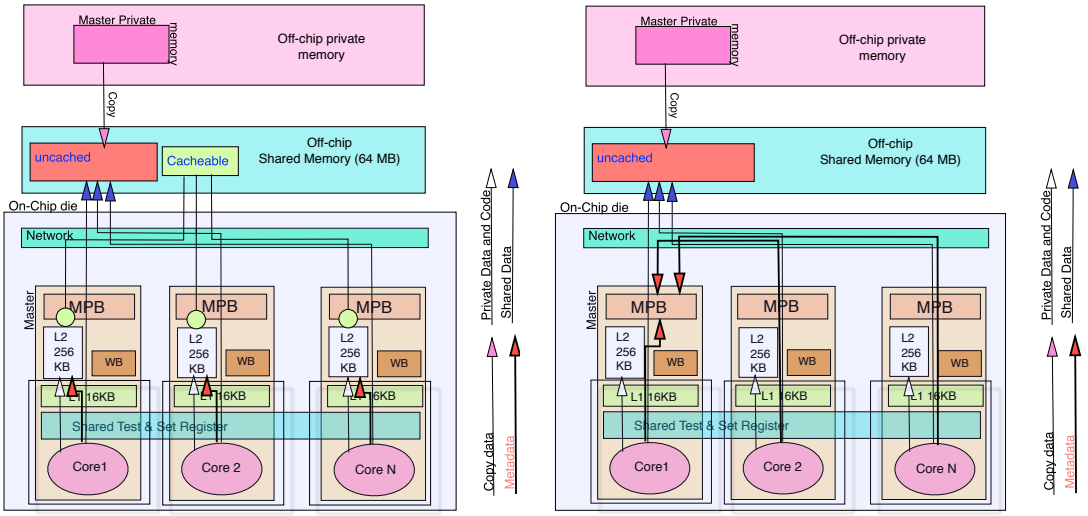
As explained in Section 3.2.1, each SCC's core has a private bank of memory (L2 local) onto which stack and private data that is by default allocated. In addition, each core has a shareable local memory (MPB) that can be used to pass data between cores. The access latency to local memory by other cores is non-uniform, since it depends on the physical distance of the core from the MPB. Moreover, the degree of contention for the shared resource and the level of congestion of the interconnection medium.

It considers the implementation that depicted in Section 4.1.4 as a baseline solution for my investigations, which referred to as **Mode 1**. Marongiu et al.[168] presented an exhaustive study of the performance that achieved by a number of possible allocation techniques in MPSoC for data sharing and metadata, to reducing the cost for data sharing. Here, the placement of metadata is considered only to reduce the overhead of fork/join implementation with synchronization optimization and different memory mode access. As a result, it shows me only the optimization in fork/join without any impact for the shared data placement. It demonstrates a several implementations of the metadata placement on top of non-uniform and explicitly managed memory hierarchies as a key to achieving low overhead cost.

In **Mode 1** as shown in Figure 4.6, the master thread ported the metadata in global shared off-chip memory in uncacheable mode. Here slave threads access shared data and metadata from the one memory controller. Since this memory bank also hosts some of the slaves (included the master) private code and data, it's expected more competition be added by other cores' activity on memory.

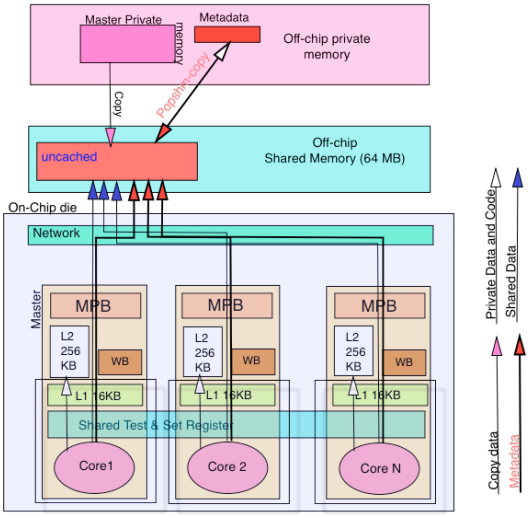
The memory hierarchy of the SCC platform is complex, and physical allocation of metadata can lead to very different performance results. Therefore, it explores a set of compiler-directed placement alternatives that take into account the memory subsystem and access mode. In the baseline implementation (**Mode 1**), it used *shm_malloc/free*() functions that designed by Intel community to support dynamic memory allocation/deallocation in global and local shared memory. As shown in Figure 4.15, the major overhead cost of team INIT and CLEANUP is dynamic memory management. Hence, one needs to investigate new ways to implement those phases.

The first variant consists in exploiting the fixed-pool memory [203] allocator to allot the metadata in off-chip and on-chip memory portion in different modes. Since metadata has read-only variables with no consistency issues arise when allowing multiple mode mapping. In addition, the maximal amount of space of the metadata will take before running the application, so it can use static allocation (i.e. Arrays allocated at compile time). New memory malloc based on pool mechanism supports fast deallocation and less memory fragmentation and related synchronization between individual groups. Other optimization added to fork/join implementation using *Inline* functions (without depending on the compiler decision) approach inside GOMP_parallel_end() to provide fast execution for CLEANUP phase and avoiding the overhead of function call. **Mode 2** represents the static pool-based declaration of of metadata in uncacheable shared off-chip memory. This **Mode 2** reduces the overhead that caused by dynamic memory management due to allocate and release metadata as expected.



(a) Allocation Mode 3

(b) Allocation Mode 4



(c) Allocation Mode 5

FIGURE 6.6: Allocation Strategies of Fork/Join Parallelism

To overcome the overhead caused by the increasing number of cores accessing the metadata that allocated by using **Mode 2**, the second placement variant the runtime redirects its allocation out of the global memory (off-chip). It calls this placement scheme **Mode 3** as depicted in Figure 6.6(a). By mapping some portion of off-chip shared memory in cacheable mode (L2) and using as static pool memory, since the metadata doesn't need to be updated again. This solution allows to remove all the traffic and extra time towards the memory block in **Mode 2** due to accesses to metadata. **Mode 4** is exploiting the master core's local memory (MPB) to host the metadata, as depicted in Figure 6.6(b). Similar to the mechanism of allocation data in **Mode 2** with exploiting MPB in MPBT mode (cacheable mode (L1) only), to reduce the time access from slave cores to global off-chip and shift the traffic generated by accesses to shared data towards

a "dedicated" memory block. Unfortunately, when the number of cores increases and the code exhibits significant activity on shared flags (synchronization variables) another bottleneck arises. Where many concurrent requests are serialized on the port of the master's MPB memory. Consequently, this mode adds more congestion among the NoC. All of the discussed performance results for those modes are explained in the next section.

The last variant is using POP-SHM mechanism [123], and will be later referred to as **Mode 5** as shown in Figure 6.6(c). POP-SHM makes some of each core's private memory accessible to all cores by remapping unused entries in all LUTs (Section 3.2.1) by collaborating with OS. Intel's POP-SHM kernel extension takes care about reserving private memory and handles any request access as well. POP-SHM requires a copy of metadata into shared memory region on the master side, and a copy out of the shared memory region on the slave side. As a consequence, POP-SHM requires two copy operations to bring metadata from private memory of master thread to private memory of the slave thread.

6.3.3 Flat Overhead

Here, it surveys the effect of supporting metadata allocating into different memory layers in the hierarchy. To overcome the scaling bottlenecks and improve the performance, this section imparts substantiation the efficient implementation compiler and runtime that support metadata through ad-hoc exploitation of the memory hierarchy. As explained in Section 6.3, there are four kinds of micro-benchmarks which are implemented in a number of possible allocation models that can be used in the cluster-based many-core system. Each benchmark runs under each of the possible placement variants:

- **Mode 1** (Baseline): Data and metadata physically reside onto shared off-chip (global) memory as depicted in Section 4.1.3. Where shared variables are cloned by default from the stack of master thread's private memory and each slave core can access them from there. This configuration uses dynamic memory allocation technique that implemented by Intel and is considered as a baseline for my experiments.

- **Mode 2**: Metadata still resides onto global shared memory, where it was originally allocated from the compiled program. Metadata allocated by using new memory malloc based on the static pool structure. This is expected to reduce the overhead of allocation and free a data.

- **Mode 3**: Metadata is allocated onto the L2 cacheable segment of the shared off-chip memory based on static memory management. This is expected to significantly reduce contention on global shared memory.
- **Mode 4**: It is equivalent to Mode 2, on the contrary, metadata is placed in the master's local memory on-chip (MPB). This configuration reduces the time that needs by each slave core to fetch the shared metadata.
- **Mode 5**: It makes a copy of metadata in a shared memory region and each cores has ability to get this copy to its private memory. This approach exploits the locality of each cores and L2 cache by accessing its private memory.

All the considered allocation models for my benchmarks are summarized based on the ratio of overhead reduction and speedup. The barrier adopted for this set of experiments employs S-MSB implementation as explained in Section 6.3.2.1.

Figure 6.7 and Figure 6.11 show only the ratio (percentage) of the overhead time (CPU cycles) from the total execution of the parallel region as follows:

$$Ratio = (\frac{Overhead}{Total\_execution}) * 100 \tag{6.1}$$

While, the Figure 6.12 explains the speedup for the four micro-benchmarks for the original version and optimization versions (S-Off, S-MPB, S-L2 and *-(S-MSBO)), that computed based on the following:

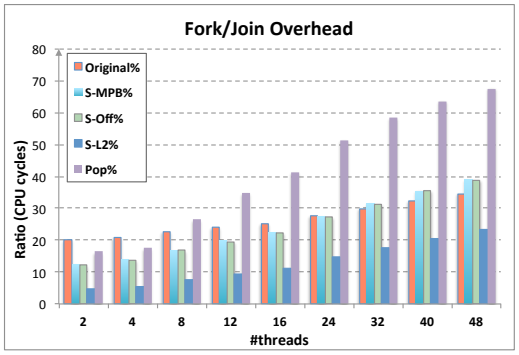
$$Speedup = \frac{Ratio_{(Original)}}{Ratio_{(Optimized)}} \tag{6.2}$$

All the curves there plotted show the scaling of the overhead time and speedup with the number of cores.

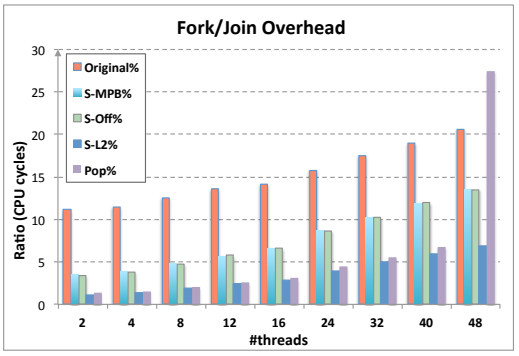
6.3.3.1 Flat Overhead without Optimizing Synchronization

This section deals with flat fork/join implementation in various memory placement and using the S-MSB primitive to control threads in parallel block.

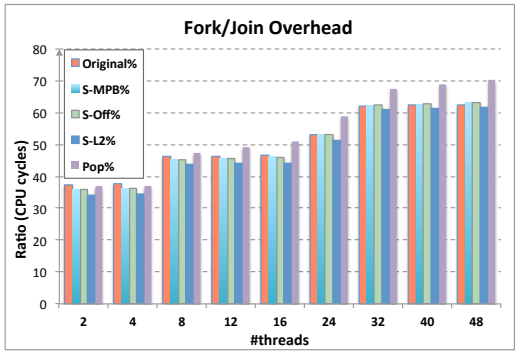
Figure 6.7 shows the overhead ratio for the various allocation modes with four kinds of micro-benchmark. Obviously, **Mode 2** (*S-Off*) and **Mode 4** (*S-MPB*) allow increasing degrees of improvement with respect to the baseline placement when number of cores <24, with the exception of load balance micro-benchmark. **Mode 5** in Figure 6.7 is referred as (*Pop*) contributes to reduce the overhead when number of cores >=8 excluding



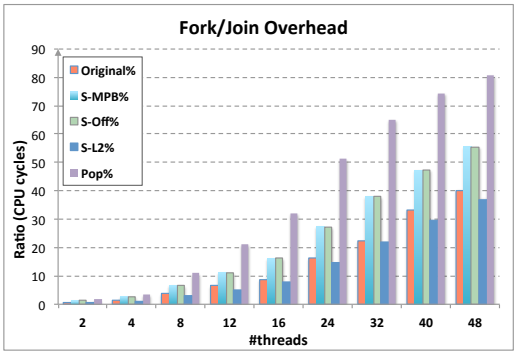
(a) Static Load parallelism



(b) Imbalance Load parallelism



(c) Dynamic Load parallelism

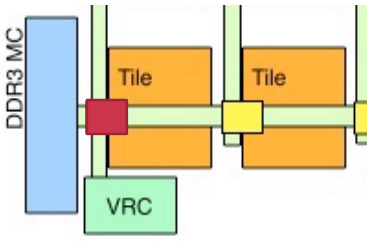


(d) Load Balance parallelism

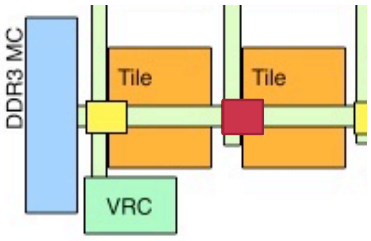
FIGURE 6.7: Overhead Ratio of Fork/Join Parallelism without synchronization optimization

the imbalance micro-benchmark. Pop added more congestion to network because many copies happened for metadata between shared and private memory regions by cores as illustrated in Figure 6.8 .

More contention arises during S-Off and S-MPB operations for most benchmarks in the network and memory port because of many requests are crowded to poll synchronization flags as shown in Figure 6.8. Where red rectangle resides in the memory controller (Figure 6.8(a)) or on the router (Figure 6.8(b)), it shows heavy traffic in the NoC.



(a) On Memory Controller



(b) On Router

FIGURE 6.8: Contention effect on Router and Memory Controller

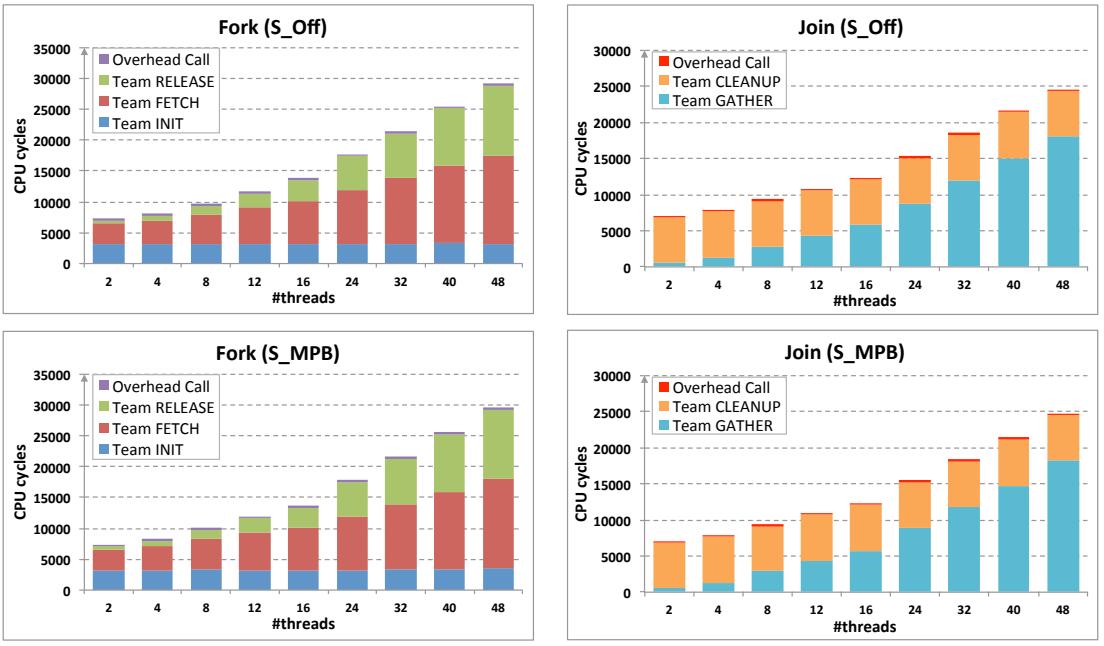


FIGURE 6.9: Cost of static load for Flat fork/join model

The bars in Figure 6.9 shows of the breakdown of the overhead that generated by S-Off and S-MPB implementations for static load micro-benchmark. As comparative of Figure 6.4 with Figure 4.15, S-Off and S-MPB reduced the overhead for INIT (in fork side) and CLEANUP (in join side) by more than 22% and 56%, respectively. While the number of cores is bigger than 24 during team FETCH, this cost increased by more than 47% for 48 threads compared to original implementation (Mode 1). In contrast, accessing to flags variables during team GATHER and RELEASE is significantly slower than in Figure 6.4 because of the access gets congested. Consequently, for processor counts up to 24 is on average faster than simply accessing metadata onto non-cacheable shared memory (Mode 1) based on dynamic memory allocation.

In imbalance load parallelism, master thread does computation more than other threads. It tried to reduce the time of gathering threads, where the master thread will be sure all threads reach the implicit barrier. Here, all modes show best scaling performance, that this happens because of this benchmark avoids the interconnect medium congestion. The bars of Figure 6.7 show that on average S-Off and S-MPB modes allow ≈34% reduction of overhead cost. Pop mode shows a surprisingly results in imbalance load by 64% reduction in the overhead when number of cores less than 48. In contract, using full number of cores in the system caused more overhead adding by 33% than the baseline (**Mode 1**) approach. For impact of dynamic load parallelism the behavior changes slightly, and in many cases **Mode 3** (*S-L2*) get stuck and performs identical to **Mode 2** and **Mode 4**.

The plot of load balance parallelism shows that S-L2 achieves ≈12% less overhead than the baseline mode. Obviously, the granularity of work in load balance benchmark depends on the number of core. Clearly, S-Off and S-MPB added more overhead than the baseline and they cannot scale with core numbers increased. Namely, the fine grain load will be finer with number of cores increasing and consequently this schema increased the overhead of Pop more than 50% based on the **Mode 1**. This behavior is due to the interconnect medium is congested when four memory banks (1 memory controller) get saturation at 20 cores [3], or both metadata and flags are accessed from master core's local memory, respectively. As expected, exploiting the L2 cache to host metadata in S-L2 mode solves the problem and achieves excellent scaling.

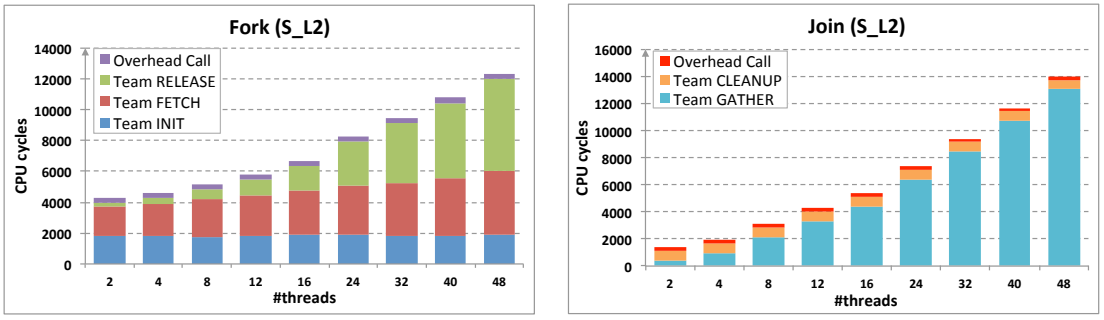


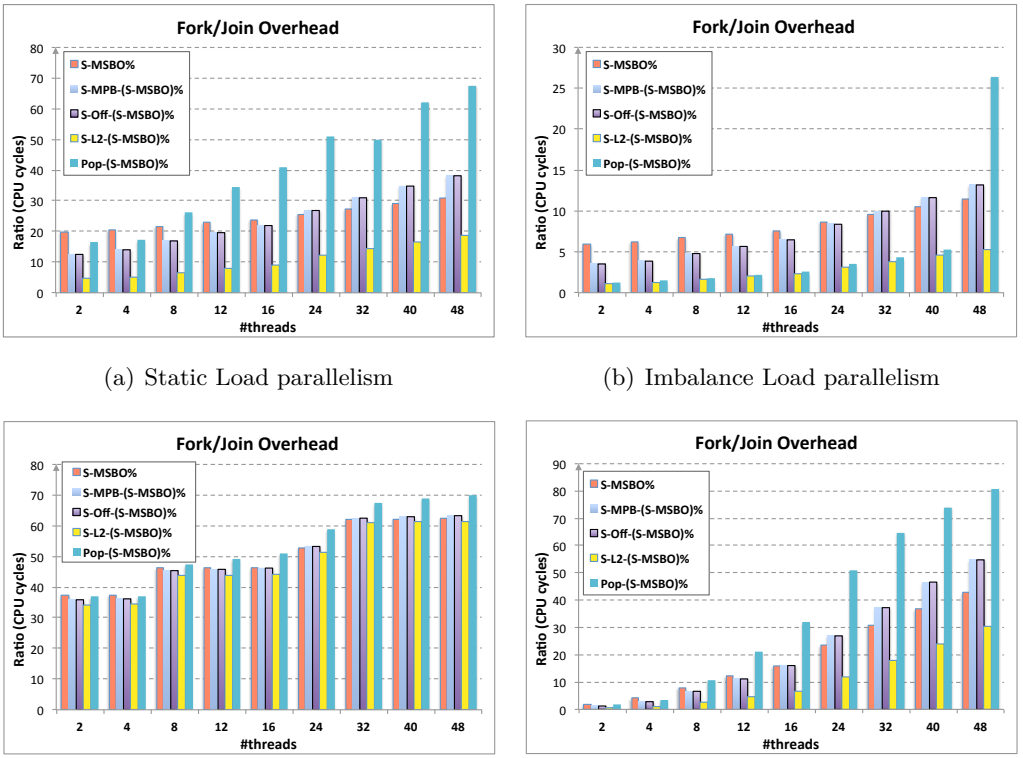
FIGURE 6.10: Cost of Flat fork/join model based on L2 cacheable off-chip memory

Focusing on all of the micro-benchmarks, it can be shown that allocating metadata onto L2-cacheable global shared memory (**Mode 3**) allows significant improvements with any number of cores as illustrated in Figure 6.7. Static load shows the overhead reduction is ≈32% for 48 processors compared to the baseline. As shown in Figure 6.10 there is no contention arises during this operation, and so neither the network nor any memory controller ever gets crammed. In **Mode 3**, the shared metadata is only accessed once at the begin of the parallel block. Furthermore, the imbalance load plot shows that the overhead of **Mode 3** allows ≈66.5% reduction. This behavior is due to the above mentioned effect of serialization of accesses in the port of the memory bank hosting metadata.

In general not all the various allocation modes allow increasing degrees of improvement with respect to baseline placement **Mode 1** because conflicting with synchronization flags, congestion in network, or no-cache coherence impact, with the exception of benchmark imbalance load computation, which shows significant differences between modes.

6.3.3.2 Flat Overhead with Optimizing Synchronization

This section studies the impact of optimizing synchronization on the various memory placement. Many synchronization algorithms are proposed in Chapter 5 to provide an efficient barrier implementation. It used the simple optimizing algorithm (S-MSBO) that reduces the time of gathering the threads in entry phase. Figure 6.11 shows the optimized version of Figure 6.7. Here, it employed the S-MSBO to do thread synchronization in fork/join model. This optimization contributes towards reducing the overhead in **Mode 1** and **Mode 3** for almost micro-benchmarks.



(a) Static Load parallelism

(b) Imbalance Load parallelism

(c) Dynamic Load parallelism

(d) Load Balance parallelism

FIGURE 6.11: Overhead Ratio of Fork/Join Parallelism with synchronization optimization

In static load parallelism, **Mode 1** has less overhead by ≈11% (for 48 threads) than the original implementation without S-MBSO primitive. Similarly, the overhead of **Mode 3** is reduced by ≈25% for 48 numbers for threads than plots in Figure 6.7. In contrast, there is no such effect in **Mode 2**, **Mode 4**, or **Mode 5**, because of the overhead of congestion that is added by accessing metadata.

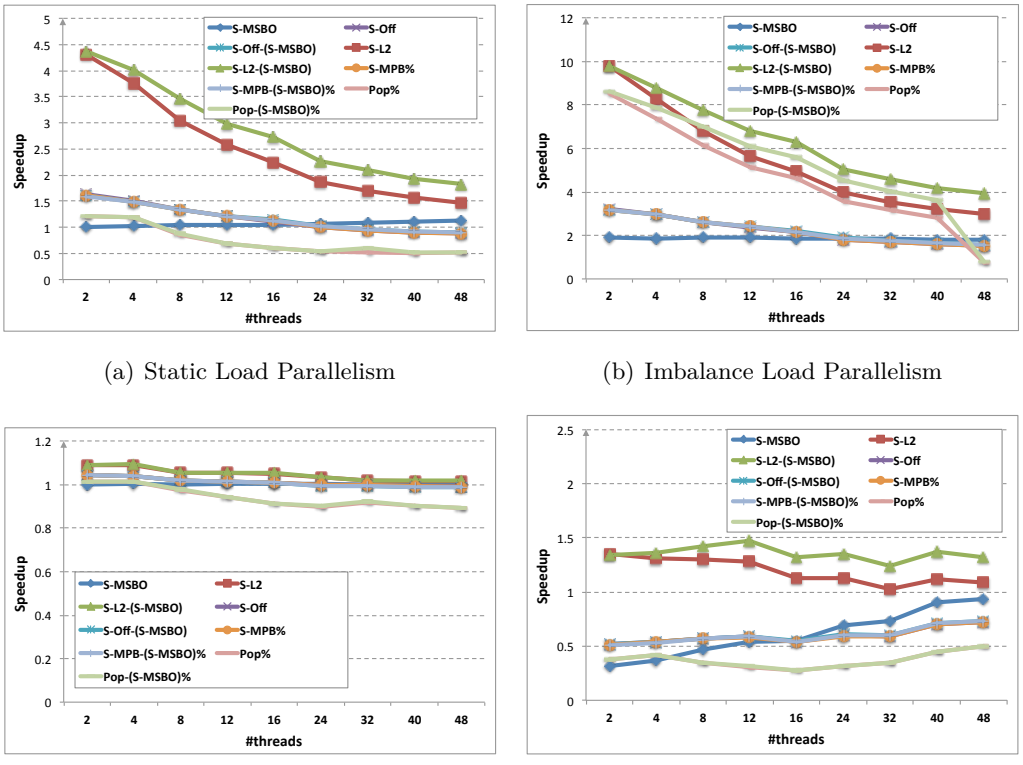
The barrier adopted for the set of experiment in imbalance load benchmark shows an increasing degree of improvement in the scaling performance with the number of cores. The baseline allocation **Mode 1** shrinks the overhead bar for 48 processors by

≈45% than using S-MSB primitives. Namely, the overhead of **Mode 1** (S-MSBO% in Figure 6.11) achieves less overhead than **Mode 2**, **Mode 4**, and **Mode 5** when the number of cores >24. **Mode 3** (S-L2-(S-MSBO)) bars for imbalance load reduces the overhead by ≈24.5% with maximum number of cores as well.

Despite the fact, using S-MSBO synchronization for **Mode 1** and **Mode 3** in dynamic load and load balance micro-benchmarks has slightly differences in the overhead with any number of cores. It will not find this influence in other modes as well for similar micro-benchmarks.

6.3.3.3 Discussion

As explained before, there are many challenges in porting OpenMP to hierarchy structure MPSoCs without cache coherence support. Regardless of the fact that OpenMP is easy to use, allows the incremental parallelization of sequential codes, and usable for MPSoC. Unanimously, the compiler and runtime need to be revisited to account the peculiarities of MPSoC hardware.



(a) Static Load Parallelism

(b) Imbalance Load Parallelism

(c) Dynamic Load Parallelism

(d) Load Balance Parallelism

FIGURE 6.12: Speedup Ratio Comparison of Fork/Join Overhead Ratio

Figure 6.12 shows the scaling of overhead time speedup with the number of cores. The speedup here is referred for a relative overhead improvement when executing micro-benchmarks with various allocation models. As illustrated in Equation 6.2, the speedup is a ratio of overhead time of the baseline allocation **Mode 1**. By this way, it can expect the maximum reduction in the overhead cost of fork/join tasks when only part of the model is improved.

This figure shows that **Mode 3** (S-L2) with or without optimizing synchronization that on average allow 2x speedup for the full number of cores used. While the other modes have a negative effect on the speedup such as **Mode 2** and **Mode 4** when the number of cores >16, and more negative impact on **Mode 5** when number of cores greater than or equal to 8.

Under those circumstances, the careful implementation of shared memory programming model on top of non-uniform and explicitly managed memory hierarchies is the key to achieving high performance. Clearly, the cost of team FETCH and RELEASE is strongly dependent on the number of threads forked or joined. Unconformity, team INIT, on the contrary, does not depend on the number of threads requested and still the overhead cost is higher. Therefore, it needs to find a new way to implement the fork/join algorithm.

6.4 Hierarchical Fork/Join

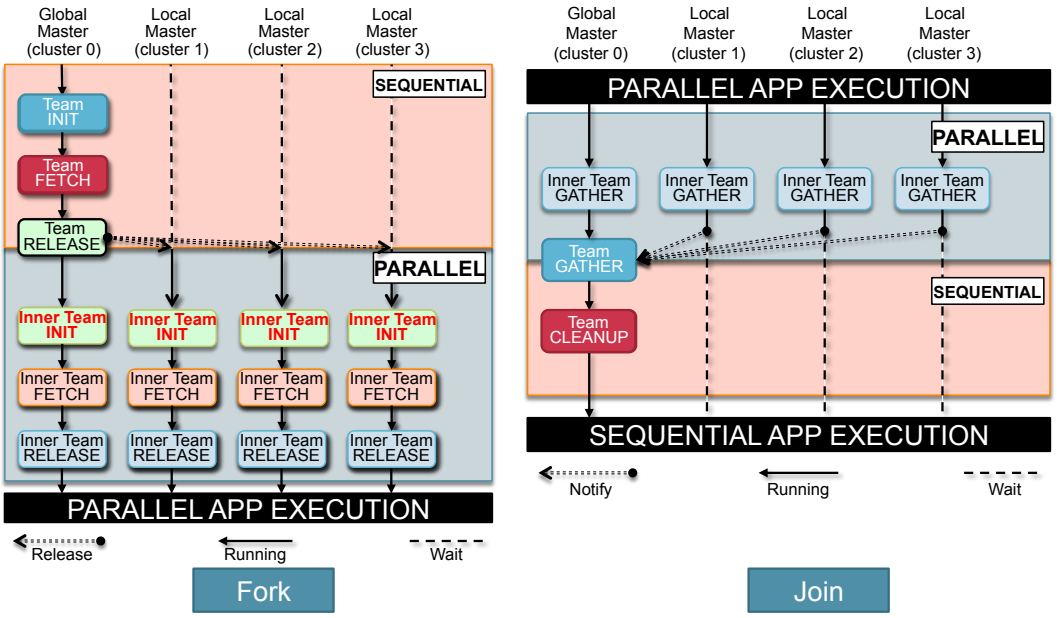


FIGURE 6.13: Thread forking and joining in Hierarchical approach

Hierarchical fork/join allows multiple threads to act as masters and concurrently collaborate to forking/joining threads as shown in Figure 6.13. This is a desirable property to address the first problem with flat fork/join as described above. In addition, if hierarchical fork/join is made architecture-aware, the NUMA effects are removed, which solves the second problem of flat fork/join. A hierarchical fork/join scheme takes into account the clustered nature of the target platform, and splits the operations in two (or more) stages. The first stage is executed only by the master thread (Global Master (**M**)), which recruits a single slave thread from every cluster. Each of these threads is promoted to the role of a local master inside its cluster (Local Master (**C1**) to Local Master (**C3**)). Local masters execute the second stage in parallel, and recruit as many slave threads as there are processors in the cluster. Hierarchical fork is implemented in Figure 6.14.

Figure 6.15 shows hierarchical join. Supporting a hierarchical tree structure allows to independently synchronize different thread teams in parallel. First, local masters (**C1** to **C3**) gather slave threads in each cluster, then the global master (**M**) at the top level gathers local masters. Note that the logical clustering considered in the fork/join algorithm does not need to match the physical clustering in the platform. The algorithm can choose for example to split the fork (and join) operation in more than two steps, hierarchically recruiting more levels of local masters to increase the parallelism of the operation. This is likely to be convenient for those architectures where the number of physical clusters is very high, while only a few processors per cluster are present. This is the case of the SCC platform. In the experimental results section, it will play with different logical cluster sizes to explore the relevance of this point.

6.5 Hierarchical implementation

This section provides more details about the implementation of *architecture-aware* (*hierarchical*) fork/join. As depicted in Figure 6.14 and Figure 6.15, When considering the hierarchical approach the slave threads are split into *local masters*, responsible for fetching processors and initial team descriptor locally on each cluster, and actual slaves. Each of the local masters (C1, C2, and C3) first notifies its availability on a private location of a global array (NFLAGS). Then they wait for the *global master* (M) thread to release them. The release signal is received via another global array (RFLAGS). To avoid the overhead associated to the contention of those arrays, it distributes the allocation of NFLAGS and RFLAGS through different memory regions. It allocated the NFLAGS array on the on-chip local memory (MPB) of the global master thread.

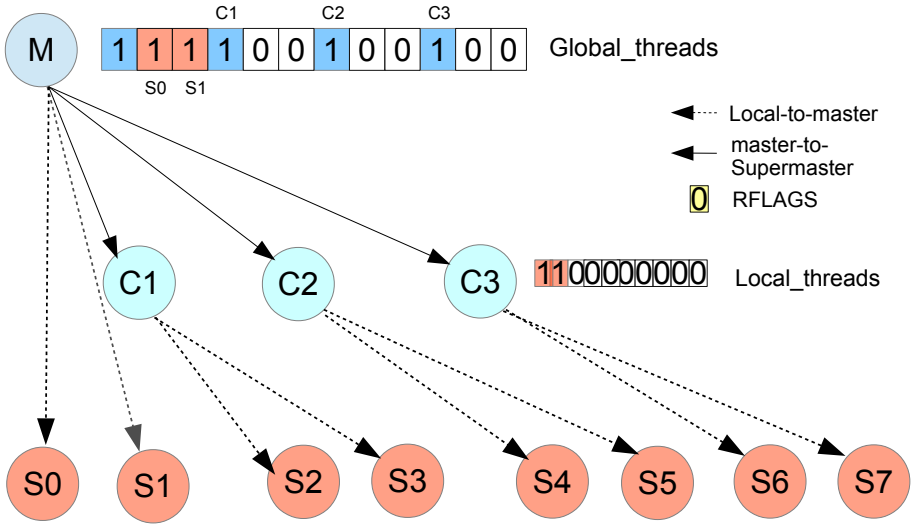


FIGURE 6.14: Nested Fork

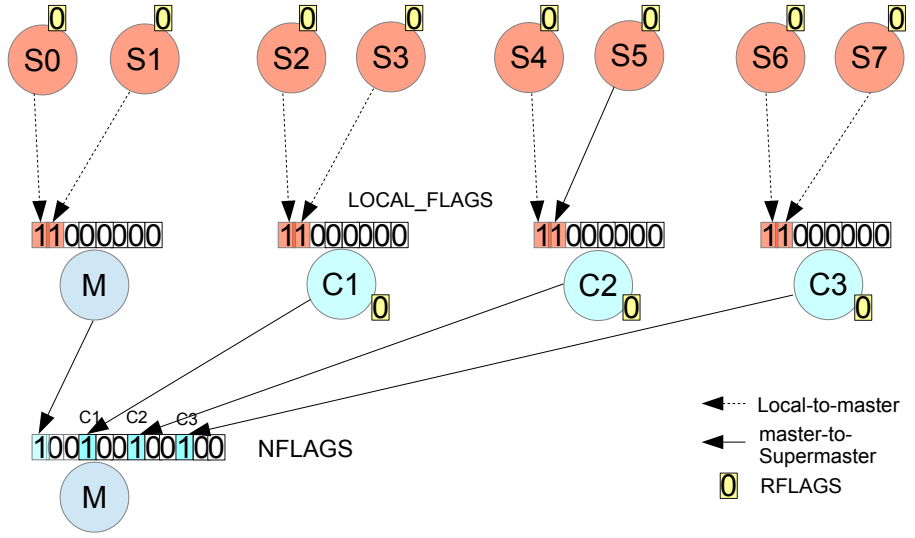


FIGURE 6.15: Nested Join

The RFLAGS elements are distributed throughout different slaves local memories. The status of local masters on the thread pool (idle/busy) is annotated in a third global array, the global pool descriptor. This array is also allocated in the local memory of the global master thread, for fast local inspection.

6.6 Performance Evaluation

My work [197] demonstrated that architectural awareness reduced the cost of fork/join (Mode 1) on the SCC and the STHORM (with large number of cores in each cluster) platform by over (2×). In similar approach, architecture-aware fork/join is achieved

by explicitly nesting in the latest OpenMP specifications [204], by explicitly nesting two `parallel` directives, the first requiring as many threads as clusters and the second requiring as many threads as processors in a cluster. The outermost regions has an associated clause (`proc_bind (spread)`) to specify that threads have to be recruited from different clusters. The innermost regions has an associated clause (`proc_bind (close)`) to specify that threads have to be recruited from the same cluster. This solution requires the programmer to be aware of the clustered nature of the architecture and the NUMA effects, and to take responsibility for explicitly coding the same hierarchical parallelization creation scheme that it has proposed. However, both for SCC and STHORM, this approach has a very similar cost to the implicit hierarchical fork/join as depicted in [197]. Where on the SCC, the explicitly approach shows reduction reduction of approximately ≈53% and ≈40% of the fork and join time, respectively compared with flat approach. While on STHORM, this approach improve the performance of fork by ≈58% and join time by ≈7%.

As a first experiment, the fork/join cost for the *flat* implementation is measured on the SCC platform, as reported in Section 6.3.3. On the SCC the flat fork/join cost of baseline (**Mode 1**) increase with the number of threads is globally less visible due to a huge cost for the INIT, FETCH (during fork) and CLEANUP (during join) stages. This is due to the fact that all the memory allocation for the data structures must be done on the main shared memory, which has a very high cost compared to the on-chip TCDM on STHORM clusters in my work [197]. This cost reduced by using custom `malloc` routines (**Mode 3**), which rely on pre-allocated memory bins and optimized for fast inspection. Therefore, it compared the performance of hierarchical approach that implemented by allocating the metadata onto the L2 cacheable portion with flat implementation in **Mode 3**.

This section evaluates the architecture-aware hierarchical fork/join. In the SCC using the physical parameters (24 clusters of 2 threads) does not lead to good results. Because of the overhead that associated with synchronization primitives and communication. Consequently, it experiments with four different logical clustering:

1. 4 clusters of 12 threads each (**4-cluster**);
2. 6 clusters of 8 threads each (**6-cluster**);
3. 8 clusters of 6 threads each (**8-cluster**);
4. 12 clusters of 4 threads each (**12-cluster**).

Figure 6.16 depicts the physical mapping of such logical clusters on the platform. Figure 6.17 shows the cost for hierarchical fork/join of 48 (max) threads on SCC using

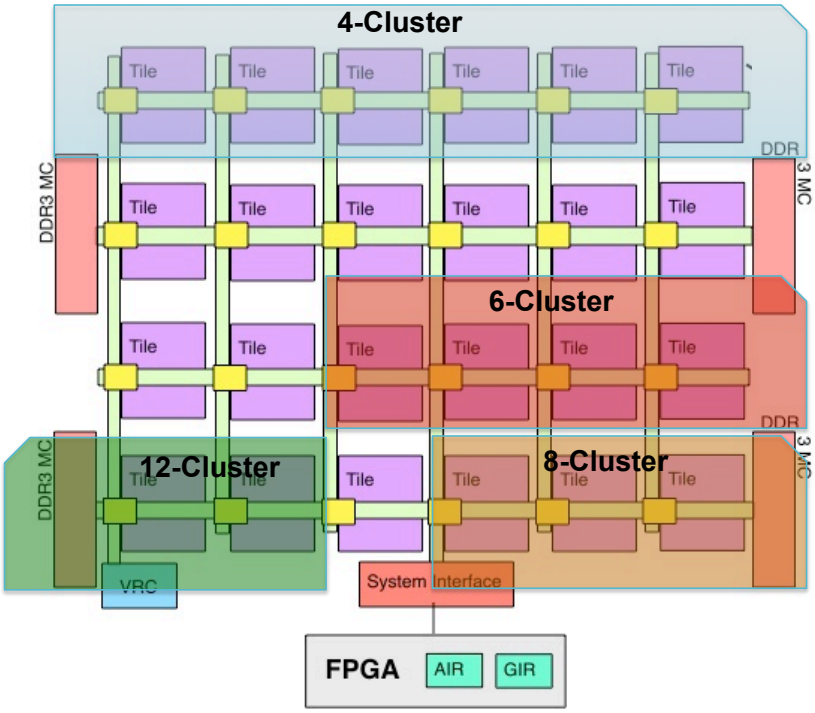


FIGURE 6.16: Abstracted cluster mapping on the SCC

the various logical clustering schemes. The plots show the time (execution cycles, Y-axis) spent on each local master (X-axis), and is broken down in the usual main phases at the first (outermost team) and second (innermost team) level. In the topmost plot, the first bar in each set illustrates the overhead of team initial (INIT), fetch (FETCH), and release (RELEASE) for the 1st/2nd-level for the fork phase. Where, each x-axis index represents the thread id of global master and local masters. The rightmost (violet) bar on each plot represents the worst-case cost, which is achieved for a mapping of the global master thread on a core that is far away from the shared memory. This figure basically covers of all the sources of the overhead that explained in Section 4.2.2, which require linearly increasing time with slaves number. Here, the static load is used only as a case study to compare its performance with the best flat implementation (**Mode 3**). Because of the optimization in this case has a similar influence on the other benchmarks.

The best clustering scheme (*6-cluster*) reduces the flat fork and join overheads by ≈12% and ≈54% receptively. The right plots in Figure 6.17 show execution time for the join phase in different cluster mapping. Using the hierarchical, architecture aware approach improves the performance by factor of 1.6, **1.9**, 1.6, and 1.5 for 4-cluster, 6-cluster, 8-cluster, and 12-cluster, respectively, compared to the *flat* model in cacheable metadata mode. In this experiment, the CLEANUP and GATHER at the innermost (2nd-level) level on the local master (0) exhibit lower cost respect to the other local masters. This is due to the alignment of support data structures (flags or global pool descriptor) to cache lines[167].

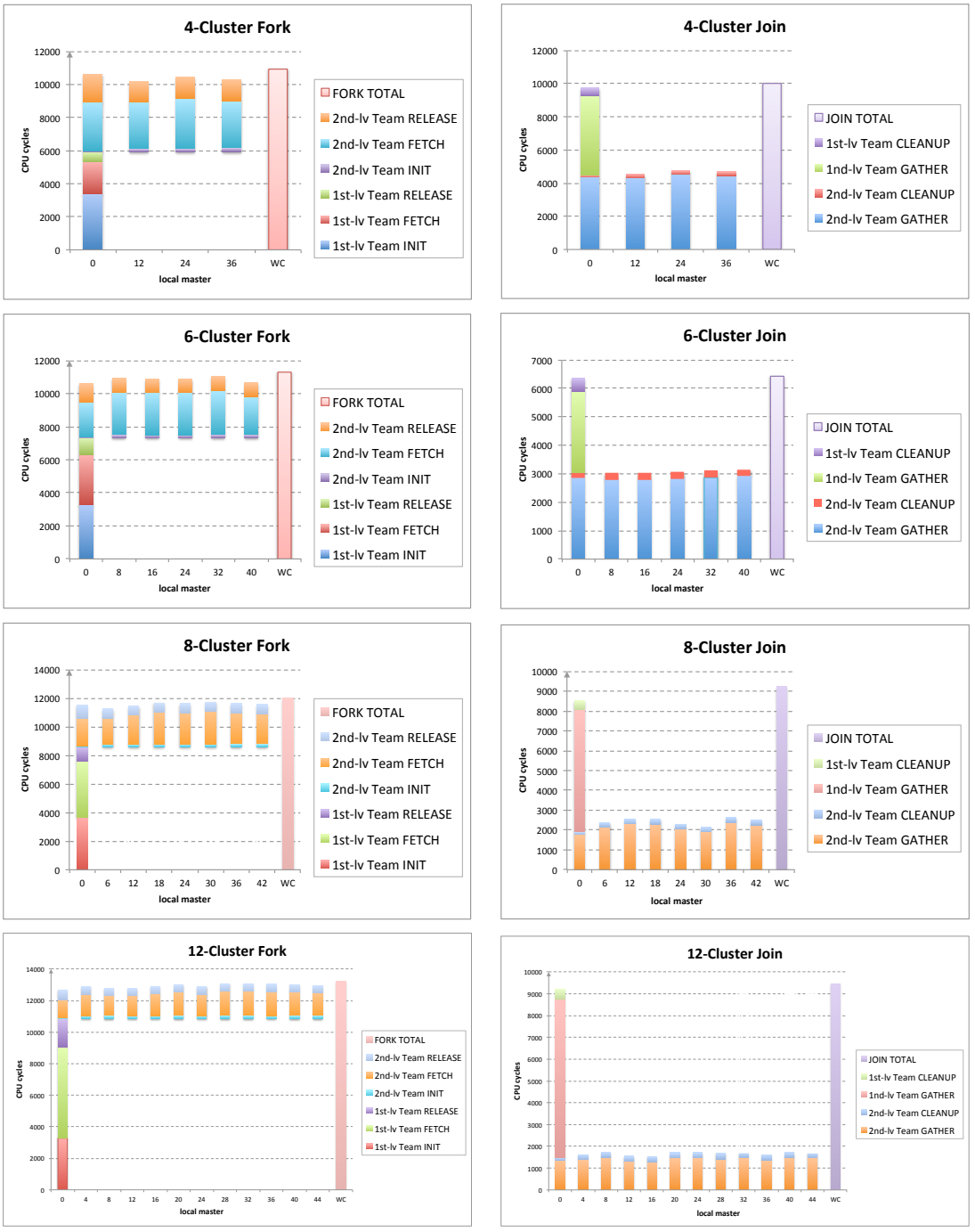


FIGURE 6.17: Cost of Hierarchy fork/join model on The SCC

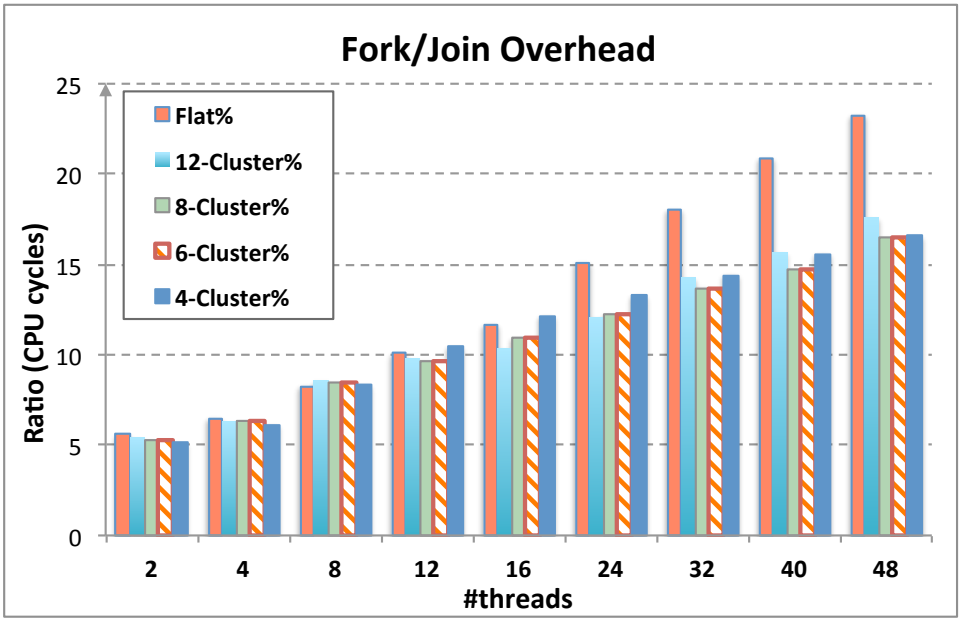


FIGURE 6.18: Overhead Ratio of Static Load

However, Figure 6.18 shows the overhead ratio for the various logical clustering with static load benchmark. Obviously, the hierarchical approach allow increasing degrees of improvement with respect to the baseline implementation (flat approach) when number of cores <12. The 12 clusters with 4 threads each in Figure 6.18 is referred as (*12-Cluster*) contributes to reduce the overhead when number of cores >=12. Consequentially, more contention arises during 12-Cluster and 4-Cluster operations in the network and memory port because of many request are crowded to poll synchronization flags. In general not all the various logical cluster mapping and with different number allow increasing degrees of improvement with respect to flat approach because conflicting with synchronization flags, congestion in network, or no-cache coherence impact. When hierarchy approach is used, it can achieve less overhead when using full number of cores and balancing between number of clusters and number of cores in fork/join mapping.

6.7 Related Work

This section talks the work related to the efficient implementation and optimization of the fork/join parallelism model with handle the issue of scalability. Many researchers have previously studied techniques to improve the performance of the fork/join execution model [47, 205–207], one of the dominant parallel programming paradigms for shared memory systems (adopted, for instance, in OpenMP and Cilk). The focus of previous research, however, is usually quite different from my research. The cited approaches to fork/join parallelism do not focus on the limitations implied by scaling to a large number

of threads, nor on the implications of NUMA communications. The resulting implementation exhibits important overheads, as the average granularity of parallel workloads there is higher than the fine-grained tasks typically deployed on manycores.

Currently, two kinds of parallel execution models have been ported to the manycore platform, which they permit coarse and fine grain parallelism. The first one is classic, *Pthread*, a lightweight thread management functions for coarse grain parallel expressions. This approach supports very flexibility on the creation of parallelism as needed, but the operations of creating, destroying, scheduling incur significant overheads. Therefore, the resulting time overheads would disallow fine-grained applications.

A similar approach to the hierarchical, architecture-aware algorithm for reducing fork/join cost is using nested parallelism. This can be achieved, for example, in OpenMP by creating a hierarchy of threads via nested parallel regions: the master thread spawns a first team of threads, then each participant becomes master to an innermost team [208, 209]. It provides a direct comparison to this approach in my paper, and it shows that the similar performance is achieved. Ojail et al. [210] recently proposed an asynchronous fork/join operations, where the fork and join primitives are shared between cores, which achieves better resource usage. Marongiu et al. [201] provide streamlined fork/join implementation for a tightly-coupled shared memory cluster using the Fixed Thread Pool (FTP) approach. My work borrows the key ideas from this implementation, but significantly extends it to a multi-cluster system with NUMA effects. The proposal here is a software-based lightweight used to point statically the threads to processing cores which directly creates a limit in load balancing.

6.8 Summary

My aim is to design an efficient implementation for fork/join model by considering the hardware privileges. During the process of designing the fork and join phases, it has been found many ways to optimize the performance of them operations. First, it consists in exploiting the memory hierarchy of the MPSoC to host private replicas of metadata. Because of metadata has ready-only variables with no consistency issues arise when using multiple copies. As illustrated in Section 6.3.2.2, several memory portions used to reduce the access time from slave processors to the master and remove almost the memory traffic during parallel regions. Therefore, Section 6.3.3 shown that **Mode 3** with a cacheable static memory approach achieved a significant improvements on the performance with any number of cores (more details in Section 6.3.3.3).

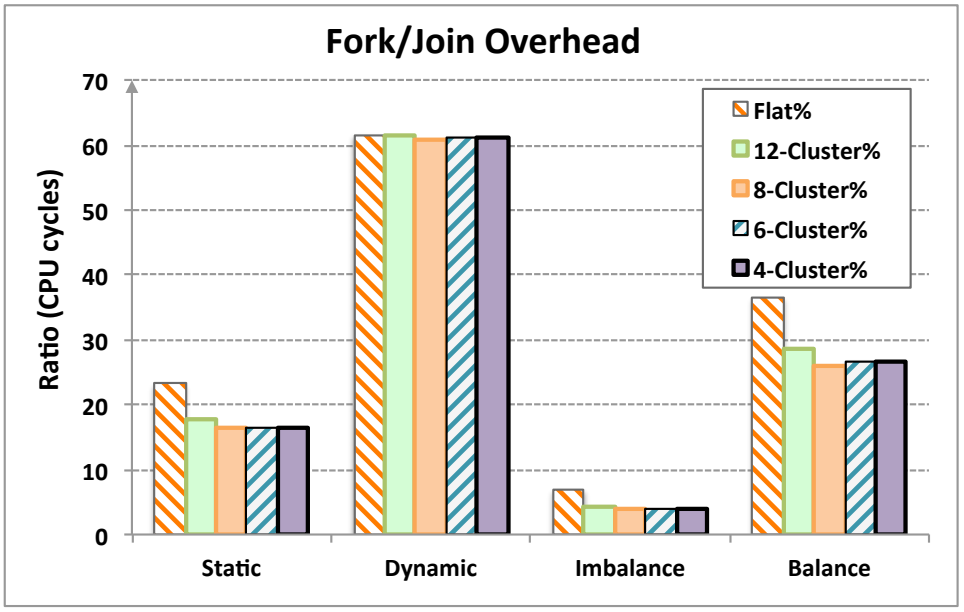


FIGURE 6.19: Overhead Ratio of several Fork/Join micro-benchmarks for 48 cores

Figure 6.19 depicts the scaling of overhead time ratio for all micro-benchmarks (Section 6.3.3) in different logical clustering and flat mappings for 48 cores only. The plot shows that hierarchy approach supports for scalable thread fork/join in large systems by considering multi-level of parallelism in a commodity many-core on-chip.

When forking and joining 48 cores with adding workload (static, imbalance, and balance), the effective execution time of hierarchy structure (4/6/8/12-Cluster) is reduced as explained in the figure. While the experiment with dynamic load shows an negligible difference in the overhead ratio than fixed length work. The variance in these times is mostly due to the tree structure, whose performance is quite dependent on the size of work that they synchronize.

However, to be able to support medium- to fine-grained parallelism, typically encountered in embedded or HPC applications, it is necessary to lower the cost for forking and joining a large number of threads. The goal of this chapter was to optimize the fork/join runtime, using an architecture-aware, hierarchical technique for thread forking and joining, which considers the physical organization of the platform in clusters. Because of architecture-agnostic sequential fork/join algorithms are not suitable for many-cores system, for two main reasons. First, laying the responsibility for recruiting a very large number of workers sequentially onto a single master thread is poorly scalable. Second, when threads are physically displaced over multiple clusters the communication underlying fork/join support is subject to NUMA effects, which increase the cost for these primitives. In addition, it has addressed these scaling issues for coordinate (spawn and join) parallel activities. By exploiting hierarchy-approach mapping for fork and join, it

has demonstrated reductions in overheads by up to ≈48% on the SCC, when creating parallel teams of up to 48 by considering spatial locality as well.

Chapter 7

Loop-Level Performance Evaluation in OpenMP

MANY-core architectures comes to allowing users not only run several applications at the same times, but also to run parallel code. Therefore, the first step of parallel code programming is pinpointing parallelism. Namely, breaking up the problem into a number of threads (is referred as a set of program instructions (e.g. a function or a loop) that issue some operations) so that each one executes simultaneously on the parallel platform with others. In many cases, the threads cannot execute independently because of the computation to be performed by one thread requires data that are produced by other threads. Consequently, dependent data must be transferred from one thread to another via communication. Apart from the fact, the parallel performance has to achieve higher demands than that of sequential program, it is harder to establish. Conventionally, programmers can achieve this goal by carefully studying the parallel machine details and discover the proper combination of machine instructions that would result in the level of desirable performance.

However, An application that incorporates parallelism in order to reduce the execution time, as a result, increasing the performance, is particularly difficult to program. Because of the correctness must prove of each individual process and also of any overt or covert interaction between them. In addition, the parallel behavior in general can not be reproduced. Based on the problem and parallel system, an application can be parallelized by using a number of patterns which are known as parallel archetypes. There are four kinds of parallel archetypes: data, pipeline, task, and streaming parallelism. This chapter introduces the performance and effectiveness of OpenMP model that can help programmers to apply the data parallelism by studying a set of different application.

To attract a lot of attention, OpenMP model used in this chapter is designed based on the *flat* approach as explained in Section 6.3.1. It used the **Mode 4** in the flat implementation to avoid the extra overhead that caused by flushing the L2 cache in the **Mode 3**, to be sure there is no data of metadata resided in L2. Two barrier synchronization primitives (S-MSB(b) and D-BTPB) are used in the flat implementation to show the impact of the barrier optimization on the performance. As summarized in Section 5.7, there is a slim difference in the performance between D-BTPB (in cacheable mode) and other barrier algorithms. Therefore, the D-BTPB barrier is chosen that has reduced the overhead by 57% greater than S-MSB(b) algorithm.

During the OpenMP translator studied, it observed most frequently mistakes which are not reported in [211]. The OpenMP compiler adds *GOMP_barrier*() to the end of the function "*main.omp_fn*" that contains the code block for #pragma parallel when the application has more than single #pragma directives. Namely, the mistake here is to be sure all threads complete them work with barrier routines, although they need no barrier since the end of a parallel region (*GOMP_parallel_end*()) is an implicit barrier by default. Unfortunately, *nowait* clause is not active in this scenario. As a consequence, the performance of parallel block will take the extra overhead to carry out the barrier. The first advice to avoid this error, the programmer needs to implement each kernel separately. Of course, this solution is not an efficient and flexible. Another solution is to replace *GOMP_barrier*() with an empty function and building your own *barrier*() that explicitly used in a parallel region.

The remainder of this chapter is organized as follows. Section 7.1 presents the data parallel paradigm as the main concept of OpenMP parallelism of many applications. Then, the design of the new OpenMP extension described in Section 7.2 to trigger flush operation as well as the compiler and runtime support. Section 7.3 documents the reduction clause implementation. The performance of OpenMP memory model reported in Section 7.4 on different access modes and frequency scaling. Finally, real-world applications selected to validate the performance of the OpenMP implementation with the proposed extension in Section 7.5 and concluded the performance scalability in Section 7.6.

7.1 Data Parallelism

One of most common parallel archetypes is data parallelism that occurs when the same operation is applied to different data. For example, when you have a lot of pixels in an image that you want to process. To getting data parallelism, you need to take that data and dividing it up among multiple processors. Data parallelism can be applied on several

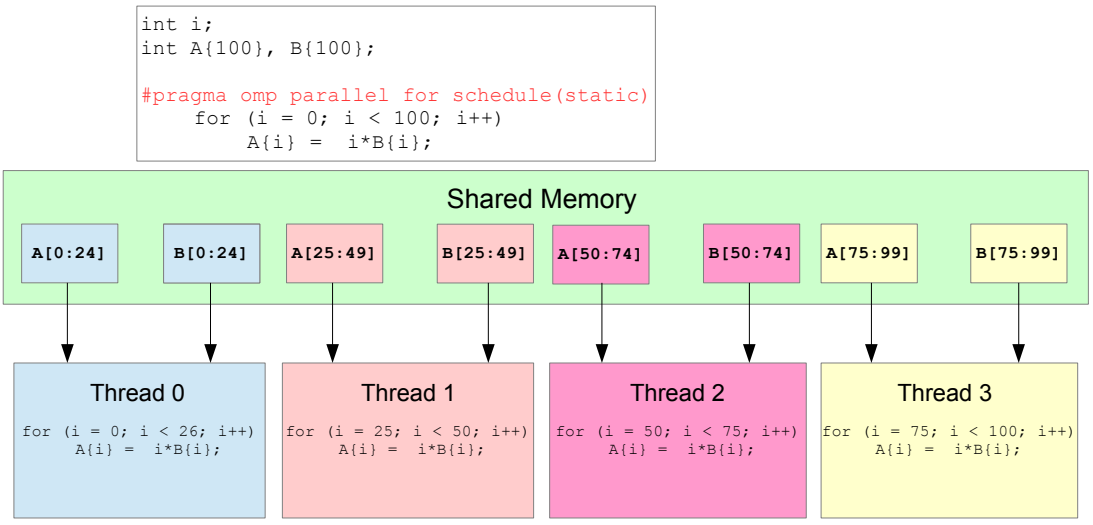


FIGURE 7.1: Example of OpenMP loop

granularities such as an *instruction level* or *loop level*. Because this thesis considers exploiting data parallelism at the loop level, data parallelism and loop Parallelization are used reciprocally throughout the thesis. Loop parallelization supported by OpenMP API and is used when there is no data dependence between one loop iteration and the next. Here, the parallelization achieved by distributing different pieces of computation data across many threads which execute the same code. Namely, all threads perform collectively on the same data set (e.g. an array), whereas each thread operates on a different partition of the set. In OpenMP, the loop iterations are distributing and scheduling over multiple threads, with just a line of code and needs little programmer intervention (here, this feature makes loop parallelization so interesting). As a result, loops with many data independent iterations can get higher performance parallelism.

Figure 7.1 shows an example of OpenMP when parallelizes a loop and assigns the execution of the loop body of a thread that spawned by OpenMP parallel directive. In this case, a special function is generated automatically by the compiler containing the code of the loop and the computation data (i.e. arrays A and B) are broken into four sub-sets which are distributed across four parallel threads. In case of nested loops, the loop-parallelism applied only on the outer loop and inner loop executed serially as part of a single iteration of the external loop.

As depicted in Figure 7.1, OpenMP allows the programmer to combine the compiler directive for the *Loop* construct and *Parallel* construct in a single line. OpenMP can control the way of distributing loop iterations and the number of iterations of the loop as well by expecting the end-user invokes the OpenMP loop scheduler. Here, the iteration is the work units that are distributed among threads as detected by *Schedule* clause. OpenMP supports different of schedule kinds for dispatching loop iterations to

the threads such as *Static*, *Dynamic*, and *Guided* with or without the specification of a chunk size (number of iterations).

Static schedule is classified as compile-time scheduling that assignments the number of iterations equally to the threads that incorporate in parallel. The OpenMP compiler transforms the loop in a way that lower and upper bounds are computed locally by each thread, based on the number of threads and on their IDs. This scheduling mechanism reduces the scheduling overhead, but the non-uniform duration of different loop iterations can lead to load imbalance issues in the system. The static approach includes many of scheduling schemes such as Block Scheduling, Cyclic Scheduling, Block-D Scheduling,and Cyclic-D Scheduling [212]. This mechanism of scheduling has played an important role of performance in the homogeneous system or the NUMA system, when the access to memory play a trivial role. Furthermore, the OpenMP API comes to achieve good data locality by using smart combination of static scheduling and chunk size. Therefore, static scheduler considered in this thesis because the OpenMP compiler used this scheduler by default to parallelize for or work-sharing. Also, it can minimize the chances of memory conflicts that arose when more than one thread is trying to access the same piece of memory.

On the other hand, the *Dynamic* and *Guided* scheduling are classified as run-time scheduling that divides the iterations dynamically as the work is being executed (run-time). The difference between *Dynamic* and *Guided* is the chunk that is progressively reduced in the size to reduce scheduling overheads at the beginning of the loop and load balancing at the end. Here, the run-time scheduling approach distributes each loop iteration to the various number of threads as they are requested. The programmer chooses the granularity at which the scheduler is invoked by specifying a chunk size. The compiler instruments the loop code by adding function calls to the OpenMP runtime that inform it that the thread is ready to execute a loop iteration. Then, the runtime routine returns continuously to the master thread to request more iterations if there are more to be executed with the values of the counter to execute the loop code. OpenMP provides this kind of schedule clause to deal with the problems in static scheduling, but it suffers from a big synchronization overhead. It is difficult to choose best scheduler for a specific parallel loop because it depends on many factors: system architecture, dependency of loop iterations and the rate of memory access [213].

As mentioned before, the loop parallelization is applied only when there is no data dependence between iterations In case of the data dependence exists, the code performs a chain of operation to enforce the computation data only flow in one direction between parallel threads and that is known as pipeline parallelism. While task parallelism uses when multiple independent code segments are run concurrently and it is mainly applied

at a coarse gain level. OpenMP supports also task parallelism by adding a specific directive (*#pragm omp task*) to a code segment. Streaming parallelism comes to exploit data and task parallelism to execute them in parallel. Here, a parallel task typically on each execution step performs in a way that reads one or more data items from one or many input streams and passed the results of computation to one or more output streams.

7.2 *noflush* Implementation

The OpenMP memory model has a weak memory model that helps the threads to cache variables and keep consistent with the memory at a synchronization point (i.e barrier). Of course, this provides room for computer system designers to experiment with a wide range of caching schemes based on the different performance and cost tradeoff. In order to improve the performance of the OpenMP implementation, the OpenMP compiler extended by adding a new clause to disable the thread's private view on the shared memory consistency. Because of "the temporary view of memory is not required to be consistent with memory at all time" [34].

OpenMP traditionally generates a code to keep the consistency view for shared data between threads. In addition, OpenMP model comes with the explicitly directive (*flush*) that helps the programmer to make things work such as implementing own spin lock. This operation used to synchronize the temporary view sequentially with the shared memory for shared variables or the variables which declared in the flush-set. However, the new directive (*noflush*) that has been proposed, it can be added to parallel directives to remove the implicit flush routine in many OpenMP constructs, instead keeping the data in the local memory (L2 cache or scratchpad) of threads.

```
int OMP_APP(int argc, char **argv)
{
int a[100], i;
#pragma omp noflush
/* a is not flushed because the flush disabled in the end of parallel */
#pragma omp parallel for private(i)
for (i = 0; i < 10; i++)
{
   a[i] = i;
}

/* a is flushed by default */
#pragma omp parallel for private(i)
for (i = 0; i < 10; i++)
{
   a[i] = a[i] + 2;
}
```

```
return a[i-3];
}
```

Listing 7.1: Example code of noflush directive

This new directive will helps the programmer to keep the data in the private cache that will be used again by the same thread as illustrated in Listing 7.1. By using this directive will disable the flush routine that is enabled by default.

In the default implementation of the OpenMP memory model, shared variable is guaranteed to be up to date when reading or writing it and without flushing it in advance. This flush implicitly written down by the compiler in an OpenMP construct. The OpenMP programmer that is aware of the application flow, can explicitly use *noflush* directive to tell the compiler to skip the default implementation of shared variables or specific variable similar to the flush directive in the traditional OpenMP memory model. In that case, *noflush* directive used to disable the consistent view of all shared variables, because the SCC does not support flush certain lines such as *CFLUSH* in x86 assembly instruction. At the end of parallel region, the compiler skips the implicit flush and restore the default setting of the memory model. Namely, the implicit flush is disabled just when call the *noflush* directive. As a result, this extension adds new optimization techniques to OpenMP model and gives expert programmers more flexibility in the software development. Furthermore, the programmer can use this extension to program a software-managed cache that automatically handles data transfers at runtime between the local memory of a core and the globally shared main memory. Thus enabling performance gains by eliminating some of the long latency of accesses to main memory.

7.3 Reduction Implementation

OpenMP API comes with the reduction clause to help the programmer to do a recursive calculation that uses mathematically associative and commutative operators to a set of data in parallel. Reduction in the OpenMP can only perform on naming scalar variables. Each core computes a partial result of the reduction operation on the variable and the OpenMP implementation collects all partial results into a total result.

To reduce the overhead of reduction clause, a temporary buffer array is used that allocated in shared memory as a message queue [41]. Each thread writes its own partial data to a specific index in this array based on the core id without any lock mechanism.

Then, the master thread at the end of the parallel region (in implicit barrier part) gathers all all partial data from from this buffer array and gets final result without a lock or atomic primitive. By this way, it reduced the overhead and waiting time for lock by running the reduction operation and avoid the contention that caused in critical directive. For future work, I would suggest to allocate this buffer array in local memory (i.e. scratchpad memory) to hide the latency of accessing this array. However, this kernel will find all possible solutions to guarantee that all threads get always the same computational load.

7.4 Bandwidth Performance Using Stream Benchmark

High-Performance Fortran (HPF)[214] is a modern version of sequential Fortran and considered as the most popular language extension in high-performance parallelism. Most of HPF offers *data parallel* that is being executed in unison by different processors on different sets of data. The OpenMP directives offers task and data parallel (more conveniently than HPF) and is seen quite similar for SPMD parallelism. In addition, it is simpler to implement than an HPF preprocessor because of mainly designed based on the shared-memory model (HPF is for distributed memory) and helps the programmer to specify parallelism directly rather than distribute the data over threads. On the negative aspect, the run-time environment has the responsibility of placing the data and keeping them consistent when the shared memory has partitions (as it usually does).

This section aims to evaluate the performance of the OpenMP implementation when accessing the shared memory in different mode access and runtime configuration. Before it investigates the memory bandwidth of OpenMP platforms on the SCC that considered as example of cluster-based many-core system. An overview of the baseline implementation of OpenMP and methodology of measurement analysis is presented.

7.4.1 Memory Model

On the cluster-based many-core system as described on Figure 2.1, there are NUMA penalties caused by different memory access costs. To better understand the selected platform and programming model, this section presents the micro-benchmark that used to characterize the bandwidth available to an application that runs on top of OpenMP Model. It first starts by presenting the memory properties of the SCC system. After that, it presents the Stream [215] benchmark and the bandwidth performance.

7.4.1.1 SCC's Memory Properties

As illustrated in Section 3.2, the SCC has diversified memory system that contains L1-cache, L2-cache, local memory on-chip (MPB), private memory off-chip, and global shared memory in different memory mode access and operations. To reach all those memory hierarchies, it needs to go through the LUT gate that is responsible to translate and redirect the memory request. Every core accesses to the private domain in this main memory (off-chip) which are distributed over the 4 available memory controller. The authors of [216] revealed that the bandwidth of the read or write access to per-core memory varies widely with the memory access pattern. That private memory is cached in cores' L2 cache. While the global shared memory mapped in not cache mode by default. Namely, each core can only have one outstanding memory request, as a consequence, uncacheable mode results in a low bandwidth for shared memory accesses especially as all cores have access the same memory controller. To activate the caching of shared memory, the programmer needs to remap the shared memory by using special devices that mentioned in Section 3.2.4. The SCC doesn't offer any coherent among cores' caches to the programmer. This coherency implemented through software methods, e.g. by flushing caches. To flush the L2 cache, a write of 0 bytes to a kernel module which acts as a device driver, it will trigger the flush routine (Section 3.2.3).

7.4.2 Stream Benchmark

Stream [215] is slowly becoming a standard to obtain the maximal memory bandwidth (in MB/s) and a corresponding computation rate for several simple vector kernels. Here, the version of Stream has been extended and allocated the source and destination arrays from the stack as static storage with a compile-time constant size. To avoid any cache influence on the results, Stream is designed as each array must be at least 4x greater than the sum of all the last-level caches.

Table 7.1 shows Stream benchmarks with OpenMP implementation and all operations are performed with double vectors. The specification of these operations are:

- **Copy** allows the programmer to discover the unceasing transfer rates between the processing unit and memory bank.
- **Scale** adds a multiplication by a scalar to the copy operation.
- **Add** verifies the memory system performance by performing multiple loads/stores.
- **Triad** merges all previous operations (copy, scale, and add). This benchmark used to stress local memory bandwidth (L2 or L1) because the arrays may be allocated

TABLE 7.1: Stream synthetic benchmark

Name	Code	Name	Code
Copy	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = a[j];*	**Triad**	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = b[j] + q*a[j];*
Scale	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = q*a[j];*	**Daxpy**	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = c[j] + q*a[j];*
Add	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = b[j] + a[j];*	**Triadplus**	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = c[j] + q*a[j] - c[j]* *+ q*b[j] + c[j] + p*a[j];*
Triad2plus	*#pragma omp parallel for* *for (j=0; j<SIZE; j++)* *c[j] = b[j] + q*a[j] - b[j] + q*c[j] + b[j] + p*a[j] - c[j] + m*a[j] + b[j]* *+ p*c[j] - c[j] + n*c[j] + a[j] + n*a[j] - b[j] + p*c[j] + a[j]* *+ n*c[j] + a[j] + q*a[j];*		

in an aligned approach such that no communication is required to perform the computation.

- **DAXPY** is a new extension from [217] that overwrites one of the input arrays instead of writing results to a third array. Namely, the extra read for array may not be required.

- **Triadplus** is a new implementation and is similar to *DAXPY*, but it has three more operations of copy, scale, and add that overwrites one of the input arrays as well. It comes to demonstrate the performance of OpenMP when the application not limited by memory speed and when the source code is compatible with SIMD operation.

- **Triad2plus** is an extension *Triadplus* to show supercomputer performance of OpenMP by using full SIMD instructions.

These benchmarks are programmed in one main program without any subroutines. Here, cores shared all the three arrays and each thread computes a chunk of the workload based on the OpenMP compiler scheduler. The size of the chunk is equal for all threads, except for the last thread that can have a larger chunk size if the number of elements

of the array are not divisible by the number of cores. The experiments used *Double* data type to be sure a write miss occurs, the cache will typically read the line from main memory and then modify the selected bytes. As a result, this implementation eliminates the possibility of data re-use (either in registers or in cache). In addition, the time and bandwidth are measured in only parallel part of the code that's shared workload over several threads without any interference from the sequential part that executes in master thread.

7.4.3 Bandwidth Evaluation

Despite the simplicity of OpenMP, many issues will influence the performance and several benchmarks designed to address them [183, 218, 219]. It is a well-known there is many scientific applications need to use the memory system efficiently because it is a key issue for the performance. There is many issues have an influence on the shared memory performance as well, such as an SMP or hierarchical NUMA system, memory distribution and differentiation in bandwidth. This section discussed several aspects of a shared memory system such as bandwidth and memory latency. In addition, this section addresses bottlenecks that result in a loss of performance at some place.

Figure 7.2 and Figure 7.3 present measurements taken on different number of cores under the default frequency setting of SCC (see Section 3.9) and generate as output the bandwidth in MB/s and latency in *second*. The memory performance measured when every core is accessing the memory simultaneously. The results in those figures show a remarkably clear distinction between four curves:

1. **SPMD** is the baseline implementation that used SPMD execution model without including the overhead of supporting a multi-threaded execution. In this approach, each thread/core works on its own piece of memory (private). The memory is allocated and initialized privately in the stack by every core. This should show optimal results in case the memory owned by the allocator or if a first touch algorithm is in place. It should be noted that every thread here works with its private memory controller according to the default setting of SCC. As a result, each thread works independently and there is no any overhead that might by OpenMP work-share directive or barrier.

2. **OpenMP** is OpenMP implementation as illustrated in Table 7.1. This approach used the flat fork/join implementation with S-MSB(b) to synchronize threads in the work-share block. This approach distributes thread access to one portion memory over four memory controller (1 portion pro memory controller) and exclude

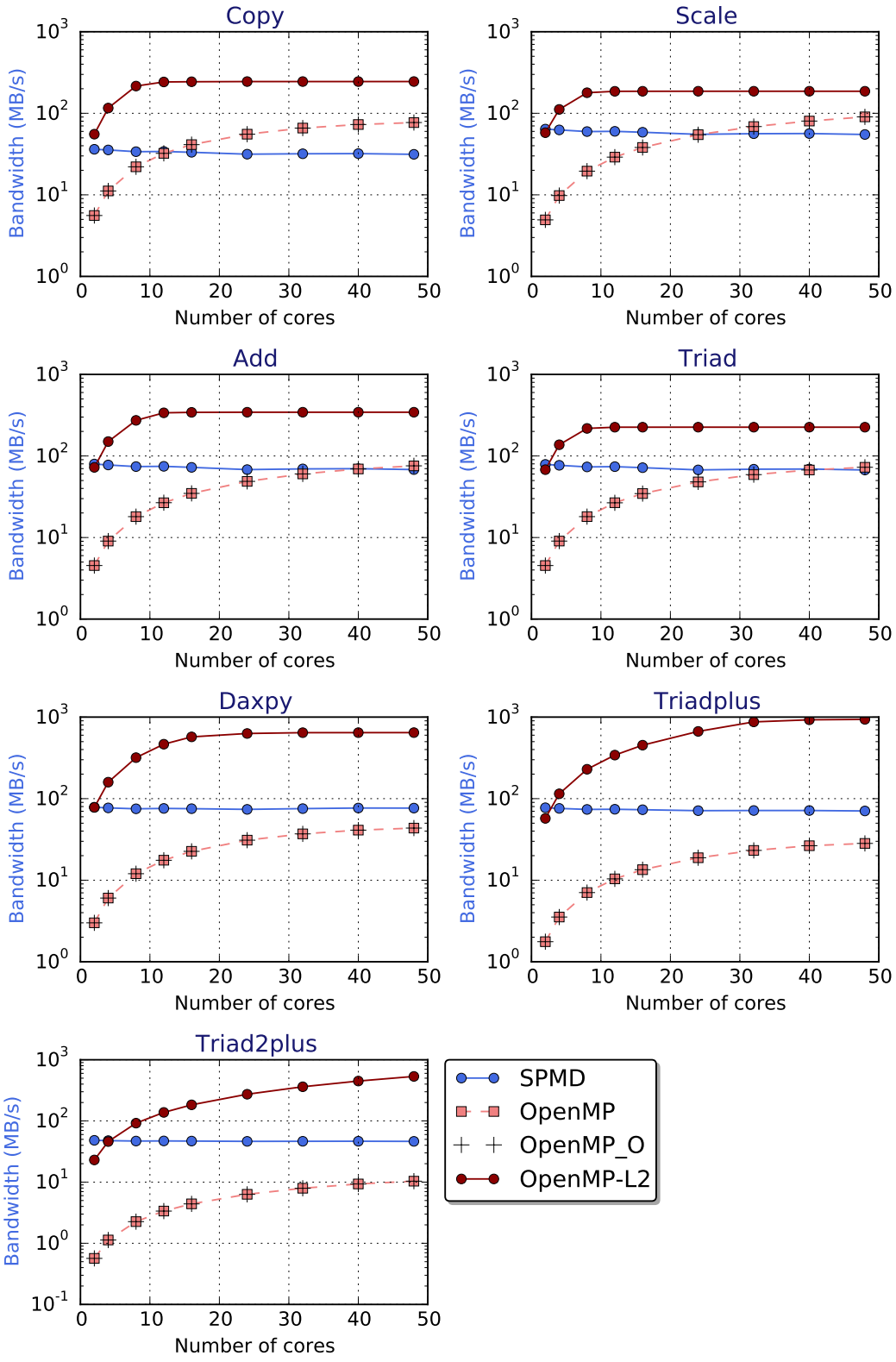


FIGURE 7.2: Memory Bandwidth of Stream Benchmarks on the SCC

the case of threads number equals to 48, here, each 12 threads used one memory controller (similar to SPMD approach).

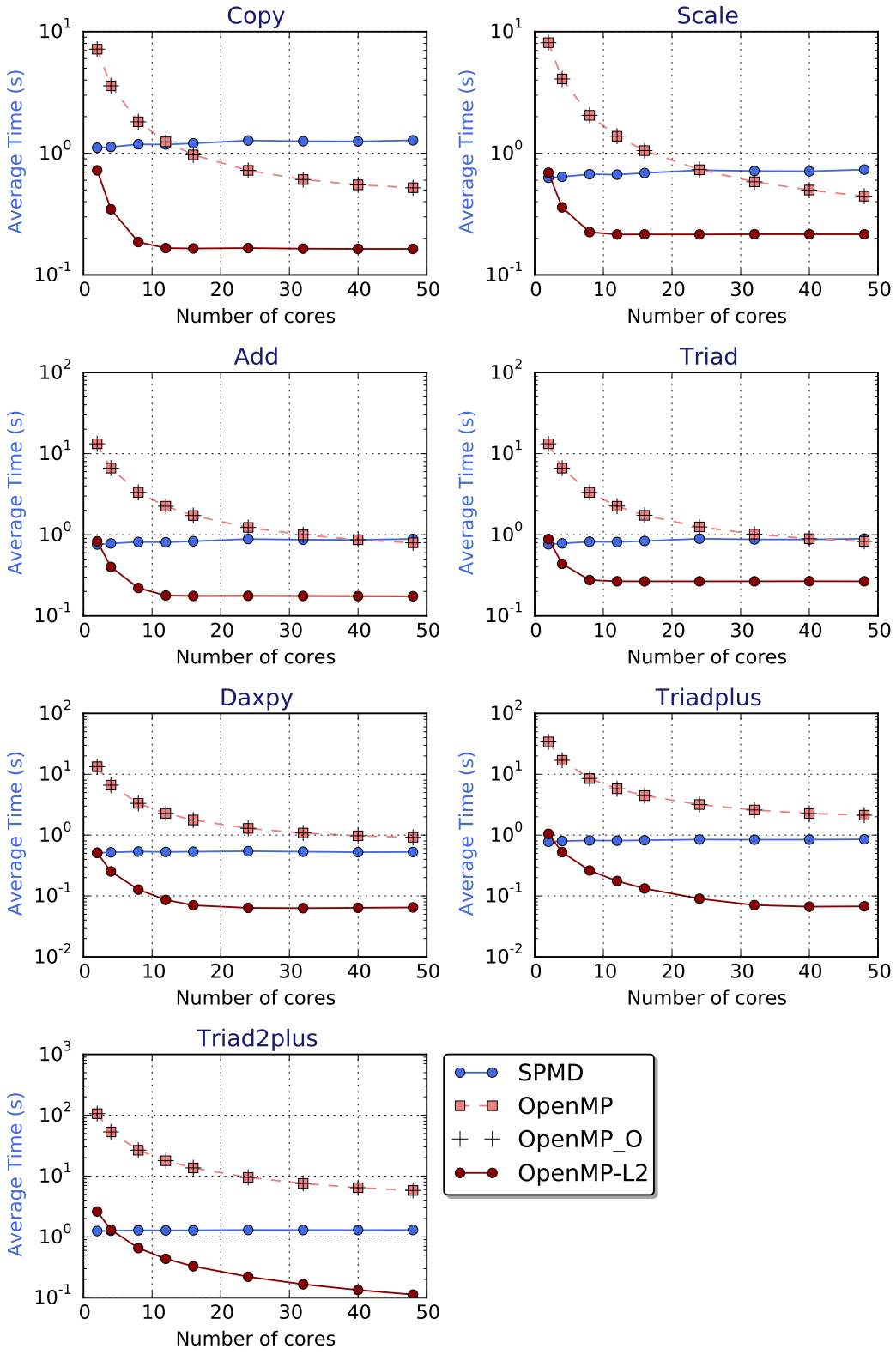


FIGURE 7.3: Memory Latencies of Stream Benchmarks on the SCC

3. **OpenMP_O** is OpenMP implementation as well that superseded S-MSB(b) algorithm by tree barrier mechanism (D-BTPB). To show the influence of barrier

optimization in the performance of memory access.

4. **OpenMP-L2** used L2 cache to reduce the latency to access the shared memory (off-chip), by remapping the shared memory that holds shared data in a cacheable L2 mode. This approach used the same implementation technique in *OpenMP* mode.

To measure the bandwidth and the memory latency, Stream benchmarks are running with several numbers of threads that chosen equal to the number of cores. The performance results represent the parallel block part (*SPMD* or *OpenMP* modes) only.

However, Figure 7.2 shows that the average bandwidth of *SPMD* has a slight difference when the number of cores continue increasing in all Stream benchmarks. In this approach, each core has serial memory access and it needs to several clock cycles delay to receiving the requested data from memory based on this formula:

$$Delay = 40F_{core} + 12nF_{mesh} + 46F_{mem} \tag{7.1}$$

Where:

- F_{core}: represents the clock cycles of the core.
- F_{mesh}: represents the clock cycles of the mesh network.
- F_{mem}: represents the clock cycles of the main memory.
- n: denotes the number of mesh network hops required to reach memory controller.

Every core continuously sends memory requests as quickly as possible. The figure shows the performance drops slightly when the number of cores is increased. This is happening because of the router mechanism such as dealing with the processing order when packets arrive at multiple ports simultaneously [220]. In addition, the bandwidth (off-chip main memory) depends strongly on the memory access pattern as explained by the authors in [216], they admitted that accessing two blocks of consecutive words will provide the higher bandwidth. As a consequence, this has an effect on the latency memory results as shown in Figure 7.3, where the latency increasing slightly as well.

In case of OpenMP implementation (*OpenMP* and *OpenMP_O*), the memory bandwidth increases with the number of cores accessing shared memory in parallel. The shared memory is distributed statically over four memory controller in not cached mode (L2 and L1 disabled). Here, the master thread is responsible to allocate the data in shared memory in round-robin fashion and controls other threads and synchronizes them

access to memory. That is mean, each core can only have one outstanding memory request, this results in a low bandwidth for shared memory accesses, especially when all cores try to access the same memory controller (accesses to the private memory are distributed over the 4 available memory controller). There is no eventuality difference in the performance between *OpenMP* and *OpenMP_O* because of the limiting factor isn't synchronized itself but general data placement/access. Furthermore, the optimization for synchronization is negligible compared to the memory activity and also the reduction in barrier overhead is tens of thousands cycles and the entire parallel block run for thousand of thousands cycles. Therefore, it doesn't show any difference between the two implementations as expected By the default the memory subsystem in *SPMD* approach is cashed on cores' L2 cache, that results in higher performance, when the computation operations increases in benchmarks (i.e: Add, Triad, Daxpy, Triadplus, and Triad2plus). On the other hand, it didn't find the influence of L2 cache in *Copy* benchmark, where OpenMP approaches achieved better bandwidth and latency when the number of cores > 12. Since every core runs an identical copy of the stream benchmark, their memory access have the same patterns and therefore they get the benefit from memory locality in terms of memory pages. For this reason, the performance increases when many cores are accessing the same memory controller, because the controller does not necessarily need to change the DRAM page for every new request. Memory latency has similar performance as depicted in Figure 7.3.

Finally, *OpenMP-L2* improves the bandwidth and latency of OpenMP approaches by remap the shared memory in L2 cacheable mode. In the SCC, the size of L2 cache is 256KB, there is no automatic management for cache coherency among all cores when activated the caching of shared memory. As a consequence, the programmer is responsible to provide this functionality through a software implementation, e.g. by flushing caches [122]. The flush operation of the L2 is expansive and it needs around 900 Kcycles to complete L2 flushing as explained Section 3.2.3. In this case, the cache needs to flush before and after each parallel block in order to make sure that no value cached in a parallel block is considered valid and will read again from the cache in the next parallel block; instead, it is forced to be actually loaded from main memory. Traditionally, OpenMP compiler inserted the *flush* implicitly in a certain position of the code when declaring shared variables as volatile. Actually, the problem is quite problematical, as each reading or writing thread needs to flush shared variables. This is one of many common mistakes which are not realized by the programmer [211]. In the *OpenMP-L2* implementation, one needs to use the flush routine twice (one to flush the sequential part, and second at the end of parallel block) in master thread. While in the slave thread, it needs to use the flush routine only one time at the end of the parallel block to be sure updating the shared dates in main memory. At the end of parallel block, the

runtime enforces the core (master or slave) to flush the L2 cache and bursts block (256 KB) of data through network and exploiting the maximum bandwidth available in the platform. While in the *SPMD* approach, the flush depends on the OS scheduling and it will flush one or more cache lines as necessary.

This approach avoids all those mistakes and provides the best performance characteristics by employing the hierarchical memory (L2 cache) that greatly simplifying the coherence issue and associated latency penalty. Figure 7.2 and Figure 7.3 evidently show the performance augmenting cache effect due to L2 enabling when every core is intensively using again its portion of the shared data. In other words, with more cores to share the workload, the size of per-core partition of the shared data set gets closer to the L2 cache size of each core, resulting in fewer cache misses that go to the off-chip DRAM (i.e. Triad, Triadplus, or Triad2plus). For instance, *Triadplus* shows a nice increase in the bandwidth going from 57.252 MB/s (1.050260 second) to 937.467 MB/s(0.067479 second).

7.4.4 Impact of Frequency Scaling

Since the different frequency of processor and network (router) has an impact on the contention behavior. Therefore, this section investigates the impact of frequency scaling on the memory performance for Stream benchmarks. Here, the influence of the contention studies on the OpenMP implementation with different processor and network frequencies. Appendix B illustrates the results of the Stream memory benchmarks with various numbers of cores. Such results obtained through the average of several executions in different frequency setting for tile, mesh, and memory by varying the number of threads from 2 to 48 cores. The results of all tables in Appendix B show the same effects of varying distances and operating frequencies of the cores, the mesh and the memory controllers.

The *SPMD* achieved the highest bandwidth with *Set3* (800/800/1066) configuration in all test cases (i.e. 48 to 39.1 MB/s in *Copy*), depending on the distance of the cores to the memory controller. As expected, the *OpenMP-L2* implementation shows a higher performance gain when using the 800/1600/1066 setting (*Set1*), for example, in *Copy* kernel is 83 to 273.8 MB/s on average. All OpenMP approaches have a similar behavior when using a higher clock frequency for the memory compared to the mesh network.

However, it has observed that doubling the mesh frequency to 1600 MHz alone does not show much performance improvement (excluding the *OpenMP-L2*) such as in *Set2* and *Set4*, especially with large numbers of cores. While when increasing the memory clock at its higher rate of 1066 MHz with the higher clock speed for the mesh as well,

it shows more improvement to indicate that the mesh is now the bottleneck regarding memory accesses.

7.4.5 Summary

It is interested to show how the bandwidth varies with the number of cores performing memory operations for *SPMD* and *OpenMP* approaches in the nature of the operations, read or write. It observed an expected drop in the memory performance for increasing numbers of cores accessing a single memory controller in parallel. As the *SPMD* curve shows in the figure and tables in Appendix B, the performance degradation slightly increases when increasing numbers of cores. While the memory performance evidently increases when the number of cores greater than 2 in OpenMP implementation. Since each core runs an identical copy of the stream benchmark and the memory access patterns are the same. Therefore, they benefit from memory locality in terms of memory pages and the performance increases when many cores are accessing the same memory controller, because the controller does not necessarily have to change the DRAM page for every new request.

Unfortunately, this effect is limited, as indicated by the performance drop or not increased when the number of cores exceeds 32, because of the varying distances result in different latencies for the memory requests arriving at a memory controller.

Overall Multi-threaded programs that are written by using *OpenMP-L2* mode reveals better performance for increasing numbers of cores. The bandwidth measured by Master thread for the parallel memory access shows that for symmetric workloads with different jobs can lead to performance improvements. *OpenMP-L2* approach shows performance, increasing of up to 680.6%, 238.7%, 405%, 234.8%, 742.2%, 1227.9%, and 1053.8% for Copy, Scale, Add, Triad, Daxpy, Traidplus, and Traid2plus respectively at 48 threads compared to *SPMD* approach. Furthermore clocking the mesh network by setting the frequency at 1600 MHz alone will not result in considerable performance improvement. Well, better results can be obtained by changing the clock frequency for both mesh and memory controllers to higher frequencies.

7.5 Benchmarking Complex Applications Examples

This section demonstrates the effectiveness and performance of the OpenMP translator by studying a set of benchmarks of the most popular sites targeting shared memory parallel applications. It shows the bottlenecks and results obtained by many code kernels, selected from: Rodinia [221], NAS Parallel Benchmarks (NPB) [183], the OpenMP

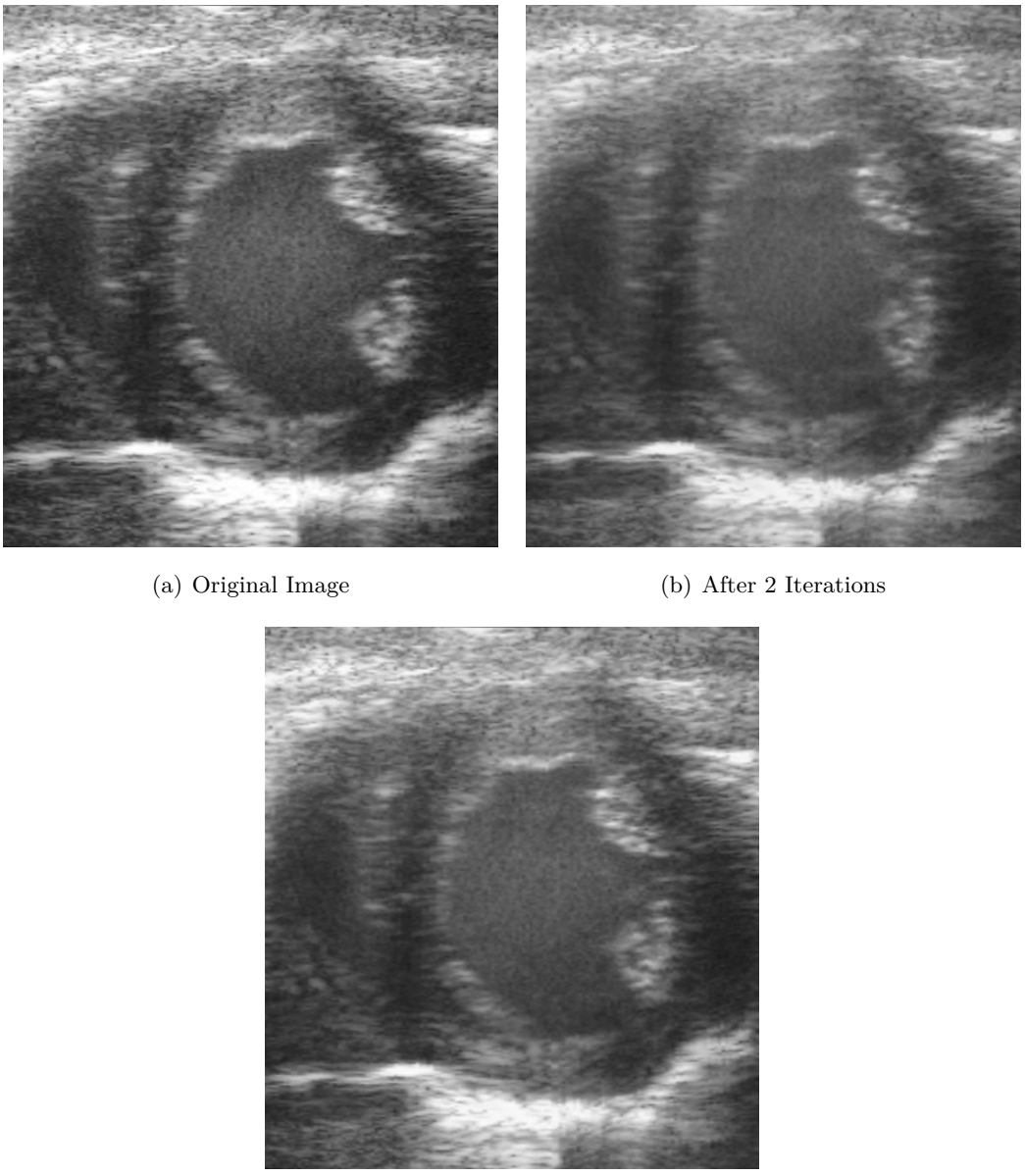
Source Code Repository [222] benchmark suite, and other real applications. All of the considered benchmarks are representative various computation patterns of the memory access patterns from the matrix and image processing (array-intensive) domain. All applications employing state-of-the-art algorithms and providing very different performance characteristics or testing different situations.

The plots in this section show the execution time of each application (parallel part only) under the above described data placement configuration (OpenMP-L2 in Section 7.4.3), normalized to the baseline. As a result, it shows the performance of the design in cache-less cluster-based system and the evaluation measured under the default system setting of SCC (see Section 3.9) and generate the time output in *second*. This time represents the actual parallel computation, plus the time spent on memory accesses without any impact for extra barrier overhead as explained in the introduction of this chapter. Many datasets of different sizes executed for each application, and the number of OpenMP threads for each test case are adjusted, to understand the OpenMP scalability. Furthermore, to discover the best OpenMP performance in each application, system, and dataset.

7.5.1 Speckle Reducing Anisotropic Diffusion (SRAD)

The first set of experiments investigates the SRAD, developed by Yu and Acton [223]. SRAD is used to remove the locally correlated noise (speckles) in an image of ultrasonic and radar imaging applications without sacrificing important image features. It exploited the nonlinear partial differential equations to tailor a diffusion algorithm [221].

SRAD kernel is iterative; it operates on the entire image in several stages sequentially in each iteration: image extraction, continuous iterations over the image, and image compression. The parallel parts of this application are: preparation, reduction, statistics, computation 1 and computation 2, which gather under one main loop, and the value of each point in the computation domain depends on its four neighboring pixels. The iteration number of this loop considers as an important fact that affect the image guilty. As shown in Figure 7.4, it shows a typical SRAD result in the different number iteration of the ultrasound image (Figure 7.4(a)). In this implementation, 2D image (Figure 7.4(a)) has been used as the dataset with 458 x 502 pixels. The iterative stage implementation classified into three main kernels which work under one main loop with 10 iterations: ROI, ICOV, and DIV. ROI (Region Of Interest) is responsible to do preparation, reduction, and statistics over the image. The instantaneous coefficient of variation (ICOV) is exploited to estimate the coefficient of variation at a single point within the image, it is known as computation 1. Computation 2 (DIV) implemented in



(a) Original Image

(b) After 2 Iterations

(c) After 10 Iterations

FIGURE 7.4: Speckle reduction using the SRAD approach

OpenMP parallel as well to calculate the divergence of the diffusion coefficients which multiplied by the directional derivatives that evaluated by computation 1 (ICOV). Computation 2 uses the divergence to update the image through computing the new pixel value.

However, it can classify SRAD as an application with data established in a structured grid [221]. Therefore, SRAD is a good representative application of a larger class of applications that it considers as a potential target for cluster-based many-core systems. Figure 7.5 shows the execution time utilization of three kernels (separate) all undergo

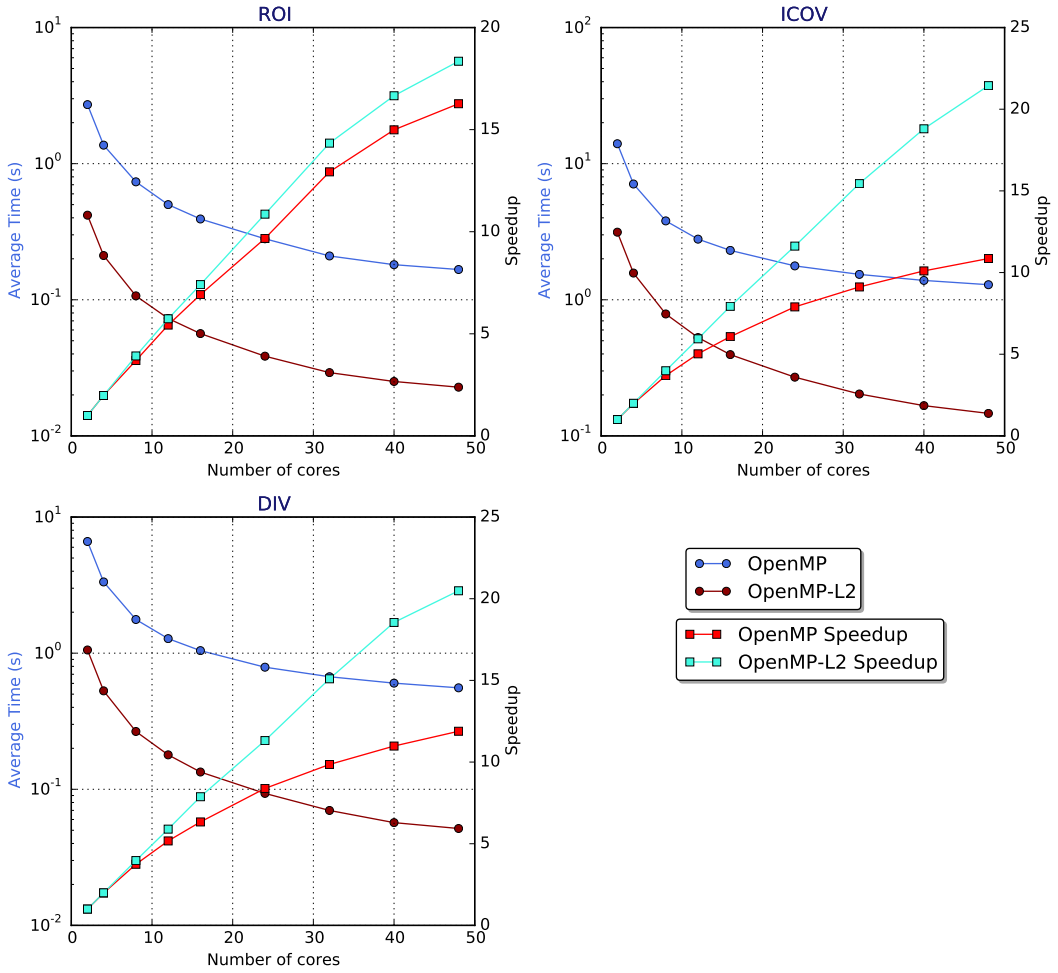


FIGURE 7.5: Performance of SRAD kernels on the SCC

a decreasing trend when the numbers of threads continue increasing. The two implementations all have a quick performance increase till they use all the core available. An interesting thing it can find is that all curves have "peak" at 48 threads, and the execution time of the *OpenMP-L2* approach keeps a significant decreasing with the number of OpenMP threads increasing, while *OpenMP* approach has a small effect, as depicted in the speedup curves. The speedup for parallel region obtained based on this formula:

$$Speedup = \frac{Time_{(P=2)}}{Time_{(P>2)}} \tag{7.2}$$

Where:

- P: is the is number of threads.

This formula will help me to understand the scalability of the system when the number of threads increased, going from 2 to 48 threads. Besides, *OpenMP-L2 speedup* shows large performance gaps as comparative with *OpenMP speedup*, using the execution

time on two threads as the baseline. Obviously, when the amount of computation in the kernel increased, it will help the application to increase the gap between two threads group as illustrated in ICOV case based on *OpenMP-L2* implementation.

In SRAD application, each thread is responsible for working with independent chunks of the image, resulting in high intra-thread data locality. Although the dependency in each element, the thread can use these elements with fewer memory accesses. Therefore, the *OpenMP-L2* approach performs well when executing SRAD application in parallel, without much performance loss. The *OpenMP-L2* implementation sees speedup because it exploits the data level parallelism in the application with fine-grained threading and efficient memory bandwidth utilization.

The two different versions of the OpenMP show speedup as expected due to their application specific construction. The speedup numbers of the *OpenMP* implementation seem relatively small because this implementation intersperses memory access instructions to complete instructions, leaving the L1 and L2 data cache idle for many cycles. As a consequence, the *OpenMP* implementation utilizes a large part of the off-chip memory bandwidth, stopping when context storage prevents more threads from being scheduled while other threads wait for their data.

7.5.2 HotSpot

Figure 7.6 shows the execution time and speedup of HotSpot application with different datasets: 64 X 64 (4K), 512 X 512 (256K), and 1024 X 1024 (1M) cells. Here, *HotSpot* is a fast thermal model suitable that is used in architectural studies. It used as a tool to simulated power measurements and estimate processor temperature based on an architectural floor plan. The power density and cooling costs going up exponentially, temperature-aware design has therefore become a necessity. *HotSpot* consists of two nested loops, where the outermost loops are usually parallelized. Those loops iteratively solve a series of differential equations for microarchitecture block. The input to the kernel are power and initial temperatures. The output of each cell in the grid represents the average temperature value of the corresponding area of the chip.

Figure 7.6 shows all OpenMP implementations perform similarly for the same dataset with different performance achieving. In the smallest dataset, each of the *OpenMP* and *OpenMP-L2* curves goes down when the numbers of OpenMP threads increasing. Here, the iteration spaces of the parallel loops are not big enough to eliminate the overhead of the long global memory access latency in case of *OpenMP-L2*. In case number of cores > 24, the *OpenMP* and *OpenMP Speedup* curves show no effect of executing the kernel code (two parallel blocks) with a large number of threads, and the time rises

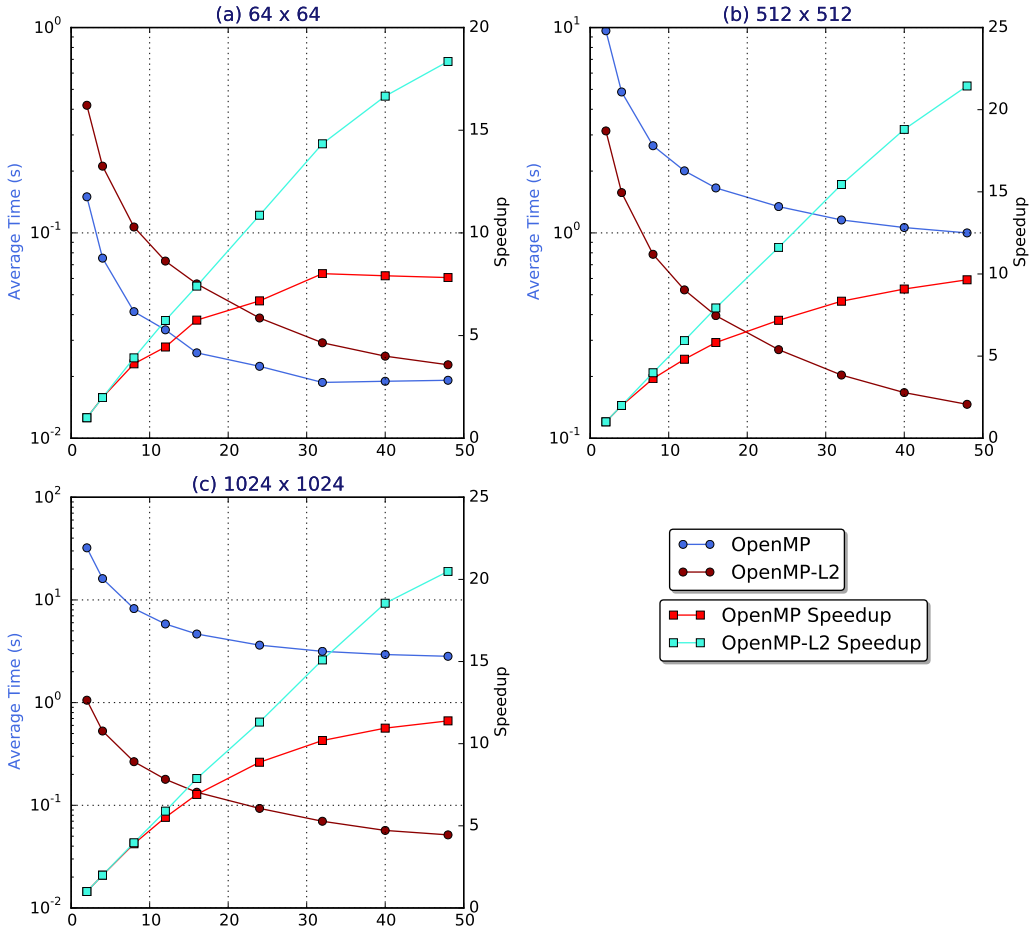


FIGURE 7.6: Performance of HotSpot kernels on the SCC

slightly after that point. Accordingly, the optimal number of OpenMP threads here is 32 threads.

For the 512 X 512 cells, the average execution time of the *OpenMP-L2* decreases proportionally when the number of threads are increased, providing the minimum value at 48 threads. The gap in the speedup and time performance of OpenMP approach with L2 enabled, increased as compared by OpenMP uncacheable mode. From this implementation onwards, the performance trends are more or less the same as those in 512 X 512 dataset size.

A point worth mentioning is the number of threads > 32 in case of 64 X 64 dataset. Using a small number of OpenMP threads can achieve a good performance with relatively small workload per thread. The overheads of synchronization between multiple threads are introduced when enlarging the number of threads. As a consequence, the execution time difference between different numbers of OpenMP threads is hidden.

7.5.3 LU-Decomposition

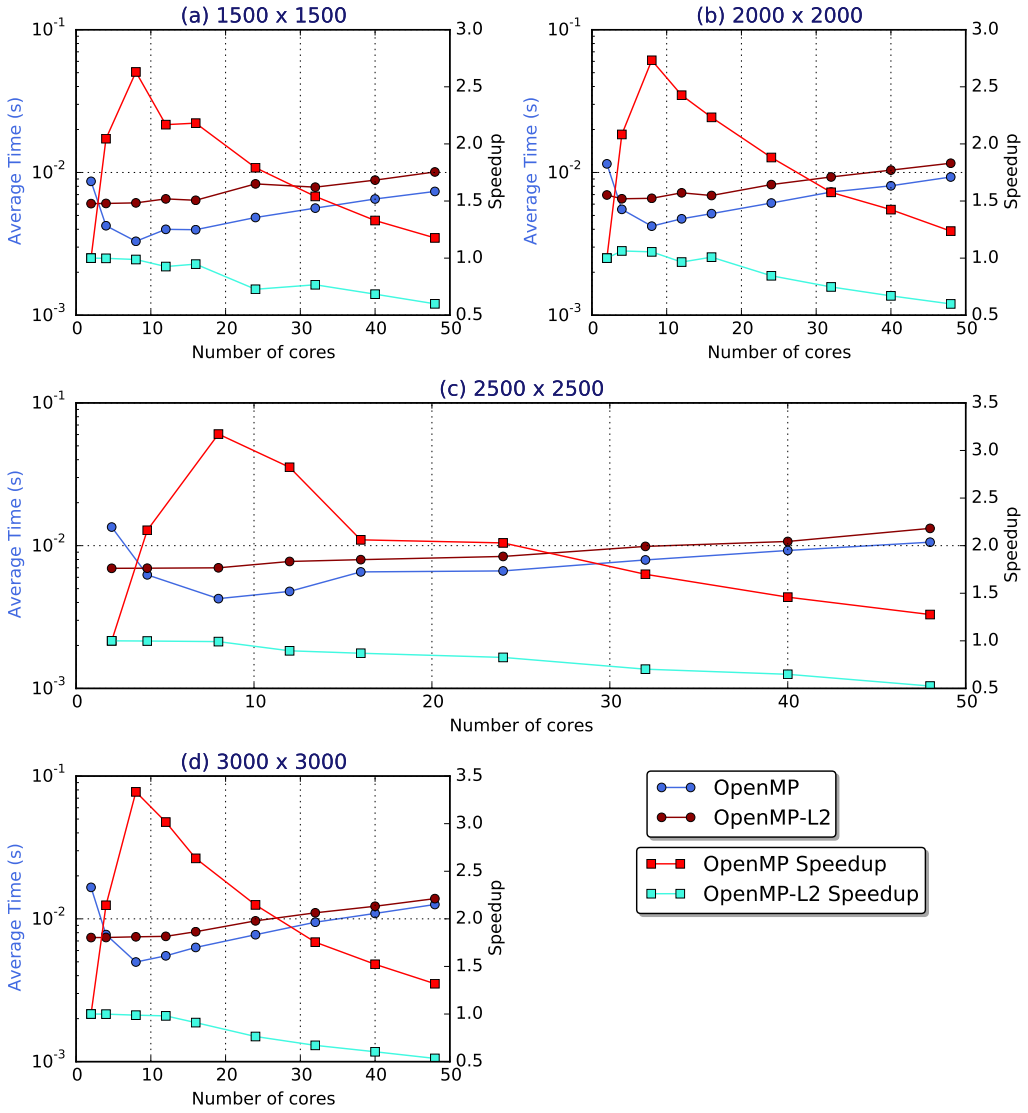


FIGURE 7.7: Performance of LU-Decomposition on the SCC

Figure 7.7 shows the results of the LU decomposition algorithm for various matrix sizes: 1500 X 1500 (8.6 MB), 2000 X 2000 (15.3 MB), 2500 X 2500 (23.8 MB), 3000 X 3000 (34.3 MB) elements. LU-Decomposition is a simple matrix decomposition tool that used to calculate the solutions of a set of linear equations. The matrix is decomposed to the product of a lower triangular and an upper triangular to achieve a triangular matrix that solves a system of linear equations easily (i.e. Gaussian Elimination). This algorithm considered as the primary way to characterize the performance of high-end parallel systems [224].

The decomposition is done in parallel by using two significant parallel regions under one main loop. Here, the matrix divided into fixed size blocks that are distributed among

threads on the system, the size of data computed by each core is based on the block size. The size of the block dealt by each thread after each iteration is not the same. As a consequence, this leads to load imbalance problem that affects performance and hence the scheduling scheme is required to reduce the load imbalance. The static scheduling is used, the chunk is divided exactly into the available threads and every thread works on the same amount of data.

Figure 7.7 observed the maximum speedup when the number of threads is equal to 8 are 2.63, 2.73, 3.17, and 3.33 (for all cases of datasets) in the *OpenMP Speedup* curve, respectively. While in the *OpenMP-L2 Speedup* curve, the maximum speedup at 4 threads is 0.99, 1.06, 0,99, and 0.99 respectively. From this result, it is clearly that increasing the parallelism by adding more threads in the application, it will be a significant increase in the cache miss ratio which will cause a spike in the required bandwidth. As shown in Figure 7.7, the average execution time curves on all cases of datasets increase highest point when using maximum number of threads. It can be seen that performance, expressed as average execution time, for *OpenMP* is consistently higher than for *OpenMP-L2* implementation. Here, it can see that *OpenMP* is a good choice for LU-Decomposition, since the memory is not the bottleneck.

Figure 7.8 shows the impact of the new directive on the performance that added to the first parallel region in the LU-Decomposition code. These results show how the new *noflush* clause (Section 7.2) could present performance improvements, according to problem size and application class used. This extension contributed to reduce the gap in average execution time between *OpenMP* and *OpenMP-L2* curves. It's possible to note that when running LU-Decomposition up to matrix size 2000 X 2000, the performance of *OpenMP* is better than in *OpenMP-L2*. From matrix size 2500 X 2500 up to 3000 X 3000, the performance trend changes and *OpenMP-L2* implementation become better and has similar behavior for *OpenMP*. Besides, the timing behavior is not very stable on the OpenMP approaches for all the datasets when the number of threads used is larger than 4.

7.5.4 PathFinder

The PathFinder is a dynamic programming technique to find the shortest path on a 2-D grid by discovering the smallest accumulated weights from the bottom row to the top row. Here, the shortest path calculation is parallelized in each iteration. Each node adds own weight to the sum that has the smallest accumulated weight of neighboring node in the previous row.

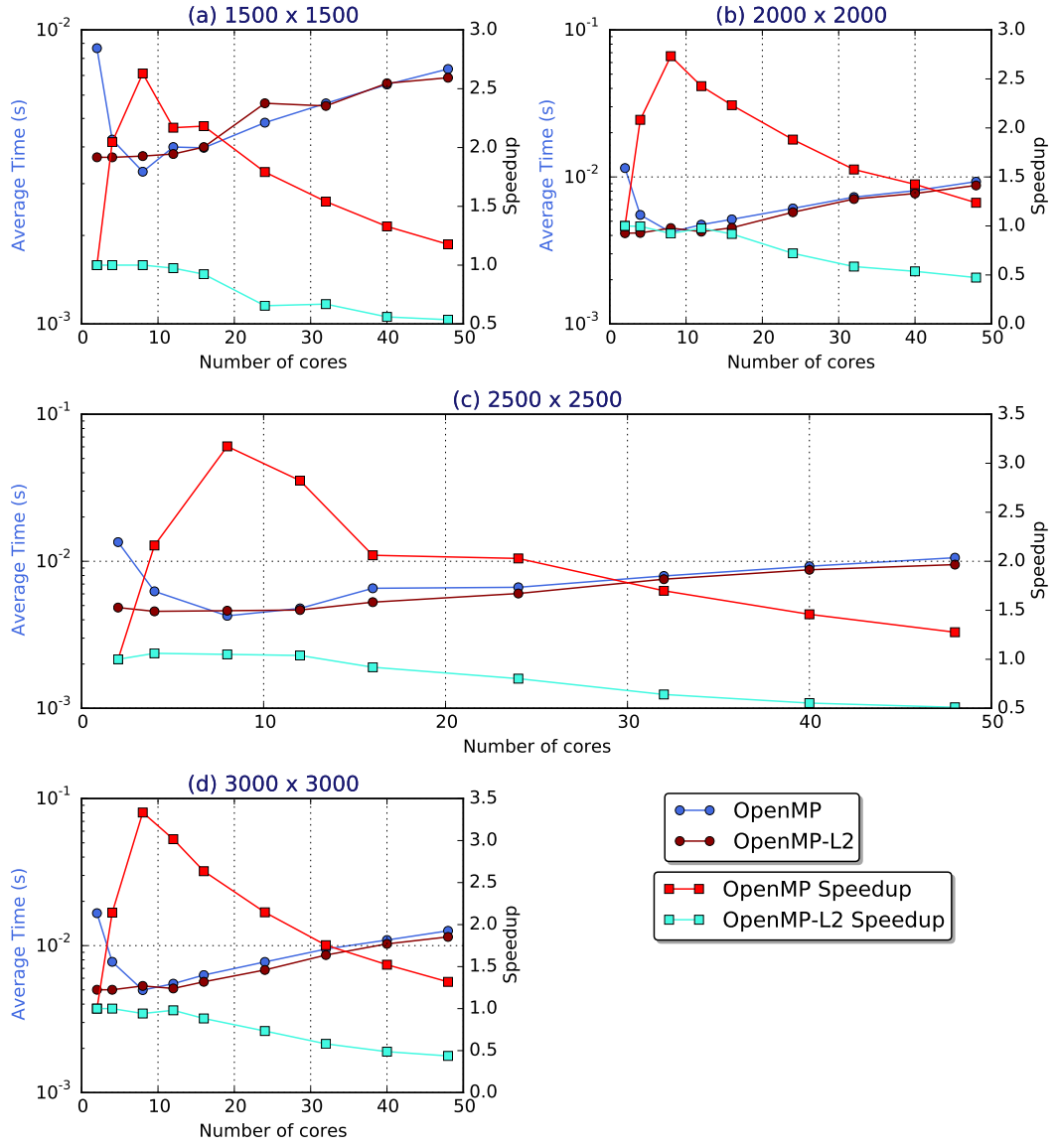


FIGURE 7.8: Performance of LU-Decomposition on the SCC after Optimization

Figure 7.9 shows the timing behavior and speedup on each implementation of OpenMP by using one grid with 100K elements (the number of columns). The *OpenMP-L2* has low execution time that drops to the minimum of 8 threads and after that a negligible increase at a large number of threads (48). On the *OpenMP*, the curve shapes are coincident with *OpenMP-L2* approach and has the lower execution time at 48 threads, but still higher the execution time of *OpenMP-L2*.

According to these results, it can see that the optimal numbers of OpenMP threads for PathFinder based on the *OpenMP-L2* are within a small number of hardware threads/cores. Because this benchmark has more memory operation and branches which bring big latency with the program.

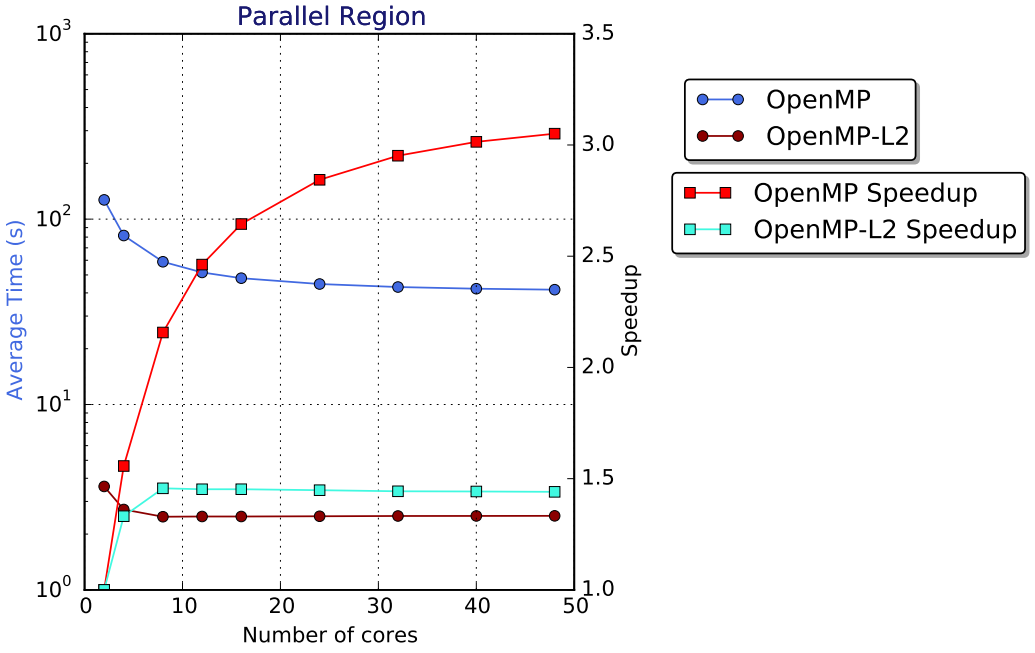


FIGURE 7.9: Performance of PathFinder on the SCC

7.5.5 N-Queen

N-Queen used to find all solutions of n-queens chessboard problem, whose aims is to place on-chess queens on a n x n chessboard so that no two queens attack each other. To solve this problem, all ways of a placing N-Queens are tried systematically on a chessboard by checking each time to see whether a solution has been obtained. As a consequence, this approach takes very large time to arrive at a solution, it therefore needs to parallelize this problem. Here, the loop parallelism technique used with effective implementation of the *reduction* clause to accumulate the solutions that find by all threads instead of using *critical* directive. The reduction technique implemented as explained in Section 7.3.

Figure 7.10 illustrates the performance of N-Queens using a variety of chessboard and implementations for four inputs: 5 (5 X 5), 6 (6 X 6), 7 (7 X 7), and 8 (8 X 8) queens (2D chessboard). Two OpenMP versions are shown: *OpenMP* and *OpenMP-L2*. The average execution time and speedup for parallel region only, are presented in a log-log plot to improve readability. In terms of OpenMP results in different datasets, it is clear that workload granularity in this benchmark and number of cores has a significant impact on the performance. *OpenMP-L2* shows the best performance in time and speedup when using the full number of cores in the system. Using a cached memory implementation with OpenMP improved the performance (average execution time) by 57.8% (5 X 5), 94% (6 X 6), 98.4% (7 X 7), and 98.7% (8 X 8) respectively, compared to the *OpenMP* approach up to 48 threads. Even more importantly, this significant reduction

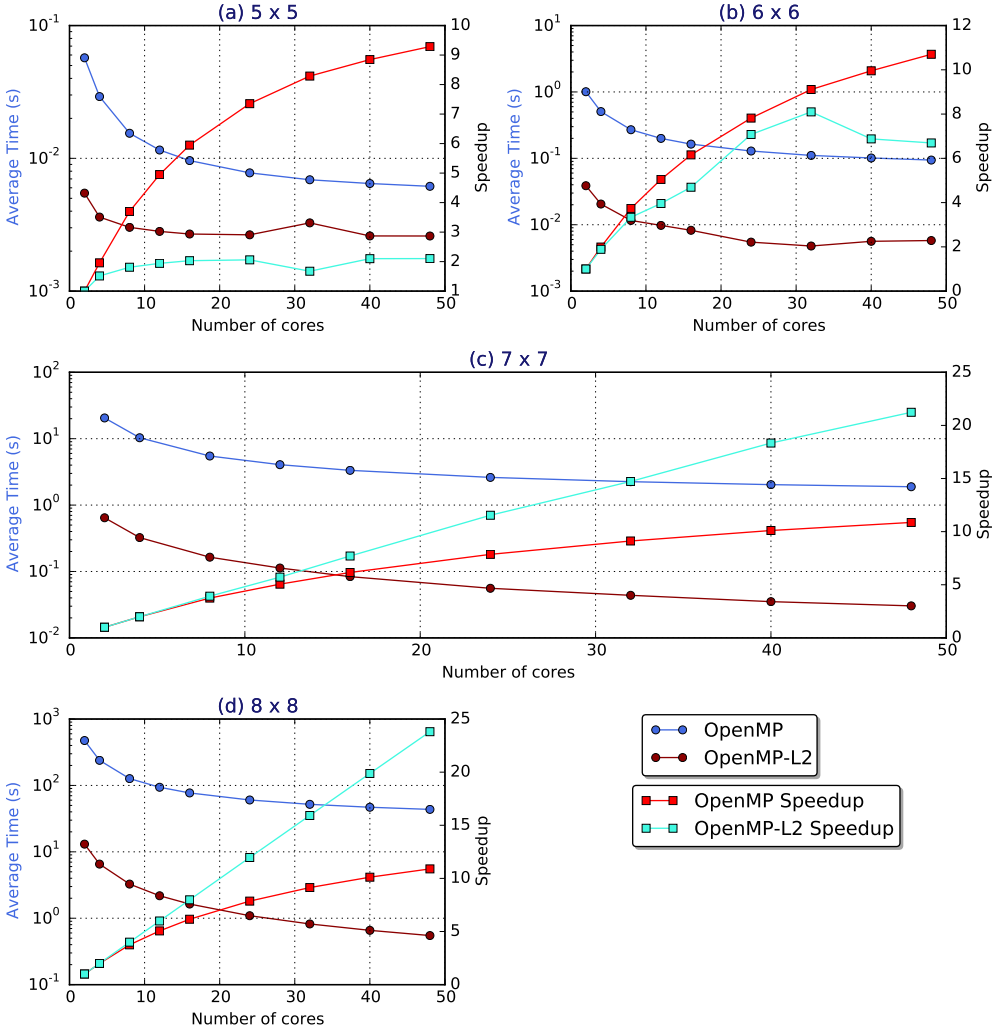


FIGURE 7.10: Performance of N-Queen application on the SCC

in execution time is achieved without negatively affecting the scalability of the program as is shown by *OpenMP-L2 Speedup* curve.

7.5.6 Mandelbrot

Figure 7.11 shows the result of the Mandelbrot set. The Mandelbrot set is a fractal structure in the complex plane that defined by the sequence:

$$z_{n+1} = z_n^2 + c \tag{7.3}$$

remains bounded.

All points outside the Mandelbrot set will escape after some number of iterations which are known to belie within the circle of radius 2 and center at the origin [225]. To

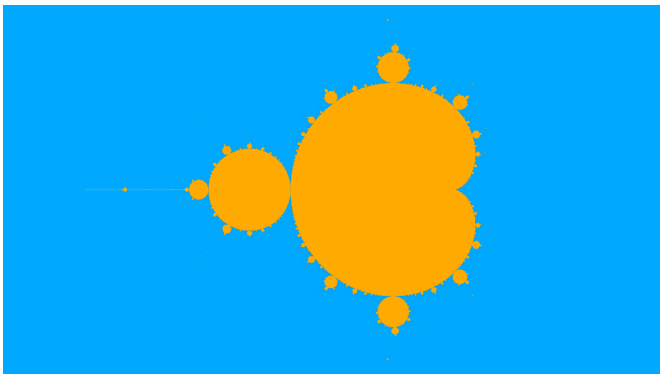


FIGURE 7.11: The output of Mandelbort benchmark

obtain a plot of the Mandelbrot set, the point C is taken to be a member of the set if the iterates never exceed 2 in magnitude for a given point. Therefore, it requires for determining the behavior of many points C under the Mandelbrot mapping to create an image of the Mandelbrot set. Mandelbrot set is the yellow shape in the middle of Figure 7.11. Because of each point can be studied independently of the others, a parallel implementation is a suitable approach for this calculation.

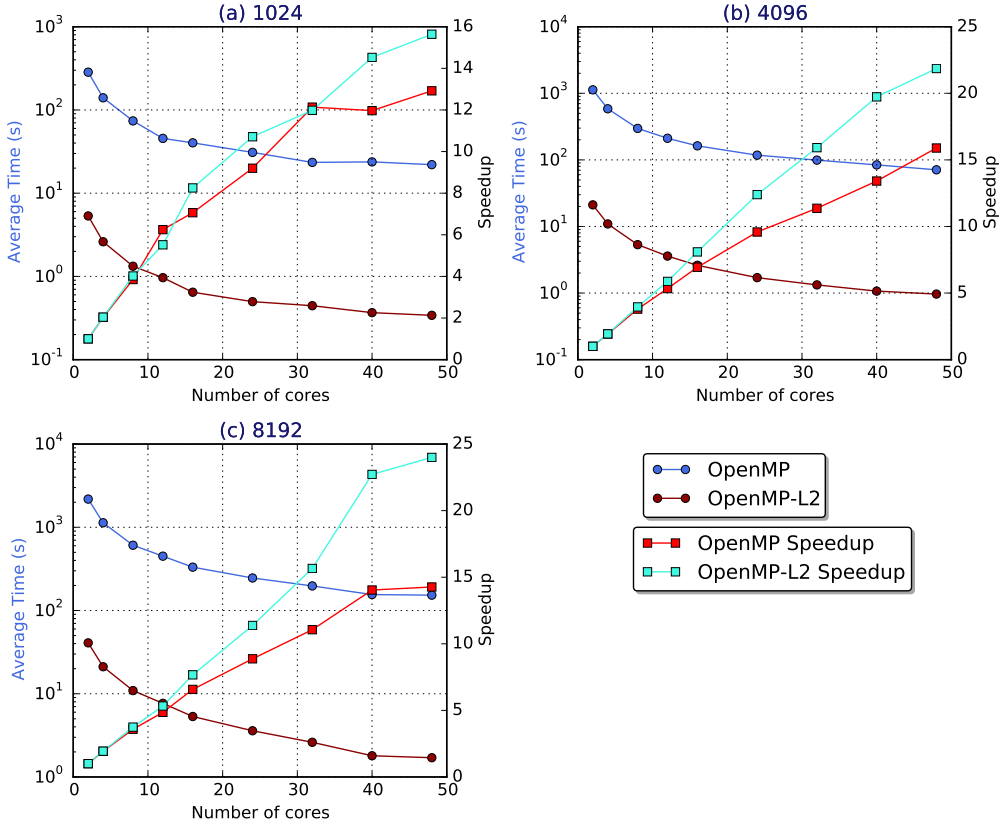


FIGURE 7.12: Performance of Mandelbrot on the SCC

This application is selected as one of the test cases in the OpenMP implementation to illustrate the parallel programming model. Because of all computations of points in

Mandelbrot area can be performed simultaneously, and this is obviously a good candidate for expressing parallelism. Regrettably, the good load balancing cannot be achieved by distributing the number of points across the n threads because the convergence can vary widely from one point to the next due to the Mandelbrot area. The parallel implementation consists of a single parallel region before main Monte Carlo iteration loop with the reduction clause that implemented based on the mechanism in Section 7.3.

Figure 7.12 shows the performance of Mandelbrot set in different grid size (the number of points to explore): 1024, 4096, and 8192, respectively, with 1000 iterations. In case of (c) 8192, Mandelbrot achieves the largest speedup with 24 with the *OpenMP-L2* approach, it is close to the linear since this application is embarrassingly parallel, and $\approx$ 15 with *OpenMP* implementation. It shows a remarkable increase in the speedup when going from 2 of 48 threads for *OpenMP* and *OpenMP-L2* approaches. This application computes intensively and does not traverse large data, therefore, there is a huge gap between running time between *OpenMP-L2* and *OpenMP* curves.

7.5.7 Helmholtz

This application used to solve a wave equation on a regular mesh using Jacobi iterative method [226]. The idea is similar in design to a map to reduce in two dimensions that calculates the equation at all points in the space and reduces on the error residuals to test for convergence. Here, the example is an OpenMP version that consists of two parallel regions with one parallel loop each with the default *static* scheduling. The program repeats one thousand iterations until the calculated value becomes smaller than a certain threshold and each thread updates a shared variable competitively to check the value satisfying the threshold. Here, the OpenMP translator replaces this code with a reduction technique (Section 7.3). The performance results of the Helmholtz application have been validated before and after loop fusion is applied.

The experimental results of the Helmholtz are shown in Figure 7.13 with two different matrix sizes of 1000 X 1000 and 1500 X 1500. Obviously, OpenMP implementation by enabling L2 cache achieved a significant time reduction in different datasets. The results in Figure 7.13 materially different from the other benchmark applications. In *OpenMP* implementation, communication overhead completely dominates the computation whenever a region accesses to the memory system. Here, this implementation shows perfect scalability speedup (*OpenMP Speedup* curve) as compared with *OpenMP-L2*. As the number of threads increases, the speedup increases due to reduction in average execution time.

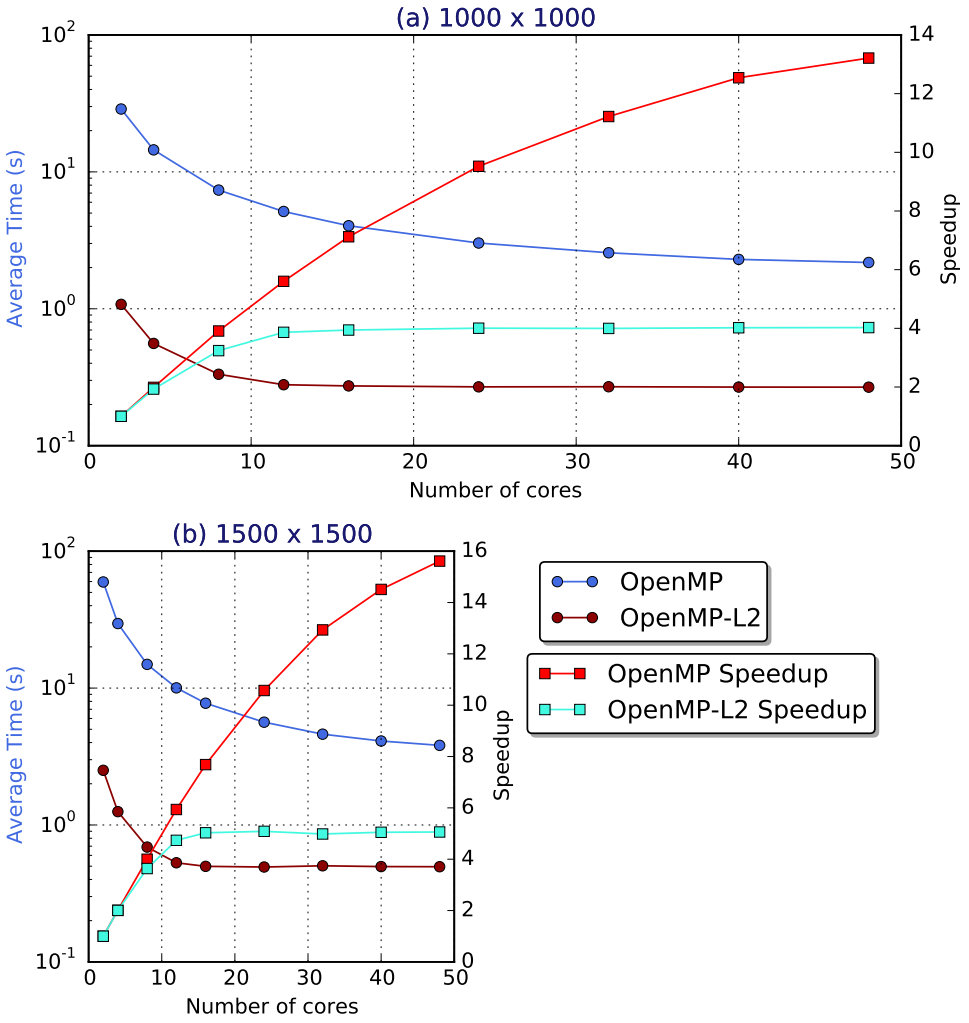


FIGURE 7.13: Performance of Helmholtz on the SCC

By using the L2 cache, *OpenMP-L2* improves the performance due to improved data locality and also slightly reduced the communication cost. As you can see from from Figure 7.13 the higher speedup achieved when the number of cores is 12 in case 'a' (100 X 100). In case of (b) 1500 X 1500, the speedup is close to linear since this application works in a number of threads <= 14. Overall, this implementation is not only improving the performance, but also contributed in reducing the power of the system by using less number of threads as compared with uncachebale mode of OpenMP.

7.5.8 Conjugate Gradient (CG)

The Conjugate Gradient (CG) is adopted from the NAS benchmarks. NAS Parallel Benchmarks [183] are widely used as a standard indicator of computer performance of parallel computers and selected benchmark is written in C. The CG method applied in many fields of application such as the area of structural mechanics and computational

fluid dynamics, oil reservoir simulation, aerospace vehicle guidance and control, circuit analysis, etc. It is used to solve the symmetric and positive definite system by iterative refinement. In addition, CG has a heavy load of computation due to the number of arithmetic operations involved in its equations. This kernel tests unstructured grid computations and communications with a matrix has randomly generated locations of entries. Therefore, the parallel implementation of CG will enhance the performance of the applications using it.

In OpenMP implementation, the parallelization achieved by creating many parallel regions to reduce the GC complexity and achieve high performance. This implementation distributes a static partition of the row-loop of the matrix-vector product among threads. OpenMP directives inserted for initializations, sparse matrix-vector product, and dot products parts of this algorithm. Based on these preparations, the performance of the conjugate gradient method has been measured using Class S to A of CG kernel.

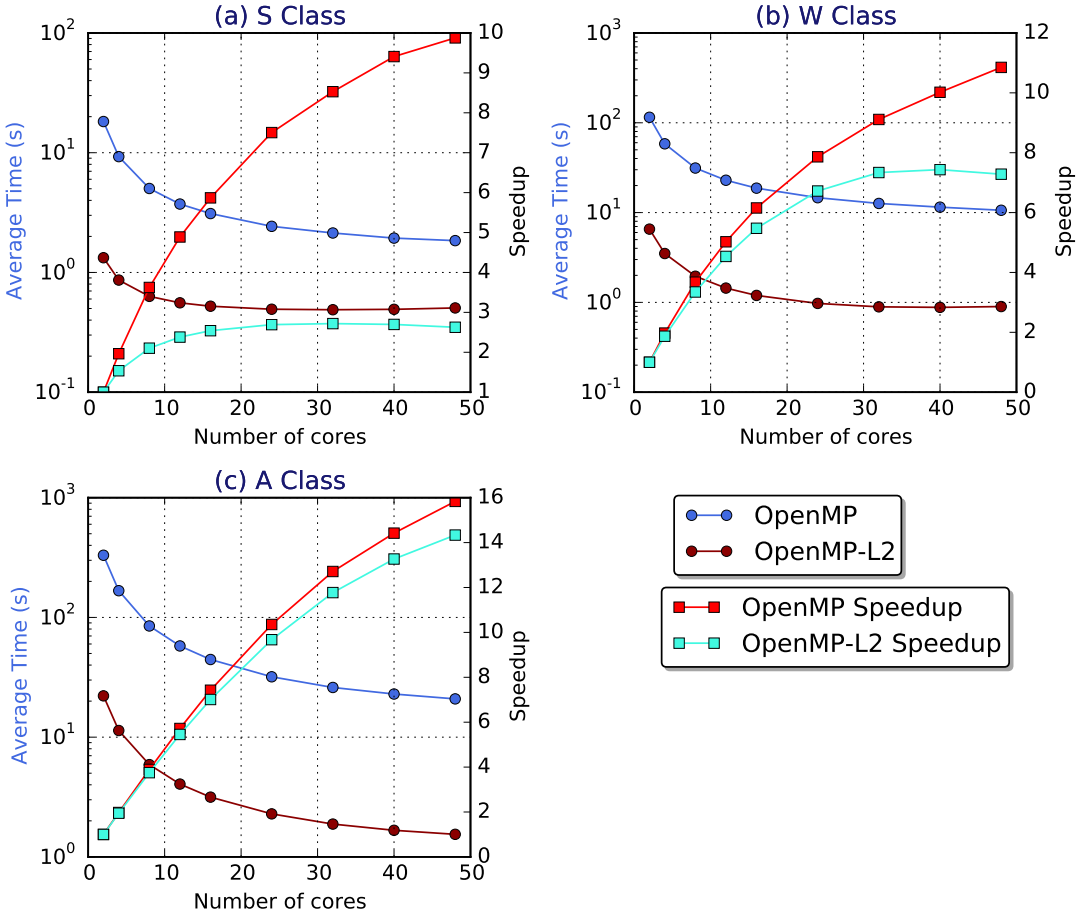


FIGURE 7.14: Performance of CG Kernel on the SCC

Figure 7.14 shows the results of three classes of datasets for CG kernel that has irregular accesses to memory. The results depict the speedup gain of parallel regions against the parallel implementation with two threads. It can see from those results, the

problem size affects the performance of the *OpenMP-L2* approach significantly in two terms time and speedup as shown in the (c) A Class results. Here, a long-lived parallel region played an important role to reduce overheads between successive sparse-matrix vector products. Furthermore, the results of *OpenMP-L2* show that the performance is improved with lower latency. The results show that there is an improvement in parallel performance of 72.6%, 91.5%, and 92.6% for S, W, and A respectively for 48 threads compared to the *OpenMP* curve.

Large datasets enable CG to scale better on both *OpenMP* and *OpenMP-L2* implementations. At 48 threads with A class dataset, there is an improvement on the speedup of approximately 445% over S class dataset on the *OpenMP-L2* approach. While the *OpenMP* approach shows an improvement of approximately 60% at 48 threads with A class over S class. It attributes this to the communication patterns in this kernel, which are long-distance and unstructured. This has a considerable impact on the performance of the CG application. It can conclude from the results, the OpenMP programming model based on L2-enable has been shown effective for parallelization of the CG benchmark. It has delivered better performance to that of the reference no cache implementation. Finally, there is another option to optimize the performance of CG implementation by adopting *noflush* clause and reprogramming the kernel.

7.5.9 Loop with Dependency

This section evaluates the performance, effectiveness, and scalability of OpenMP approaches using OmpSCR benchmark suite [222]. Figure 7.15 shows the performance of a set of variants of the *Loop with dependencies* benchmark from OmpSCR repository. This application has a number of loop parallelism schemes which are considered to resolve loop with forward and backward carried dependences. This application provides several interesting case studies, representative of real application patterns. As shown in Figure 7.15, the application includes the bad parallelized codes for two loops with four proposed solutions. One of the solutions has loop parallelism by building a threads virtual pipeline. This application nevertheless quite useful to demonstrate and experiment with different OpenMP programming strategies for non-trivial loop parallelization.

However, it shows a remarkable increase in the runtime time when going from 2 to 48 threads for all of the benchmarks. Especially for *loopAso1* (Loop A solution 1), the average execution time increases by 131.7%. Because this solution eliminates the dependences of *loopAbad* by duplicating data and distributes the inner loop into two separate loops. As expected, the overhead will increase by adding more parallel region.

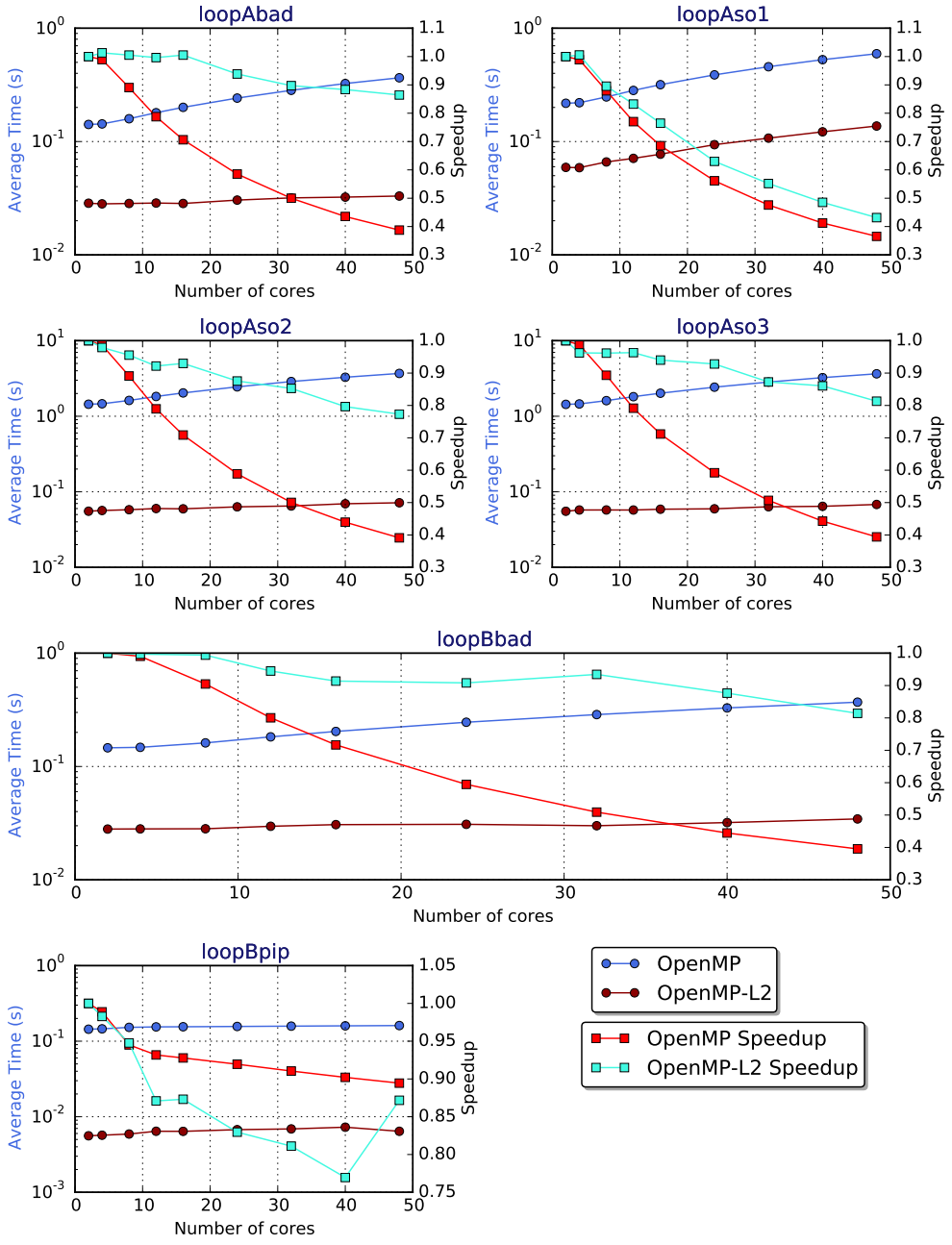


FIGURE 7.15: Performance of Loop with Dependency kernels on the SCC

But still the *OpenMP-L2* approach needs 77% less time (at 48 threads) to complete the loop comparative with *OpenMP* implementation.

To optimize this implementation, the *noflush* clause (Section 7.2) used to enforce the parallel block do not flush its data in L2 cache. Figure 7.16 shows that this optimization (*OpenMP-L2-O*) reduced the execution time by 40.5% and 23% at 2 and 48 thread respectively. The speedup changed by using this optimization as well. As a result, the better cache utilization on OpenMP lead to this difference in the performance. Overall, the *OpenMP-L2* approach has the best performance in terms of time in all kernels.

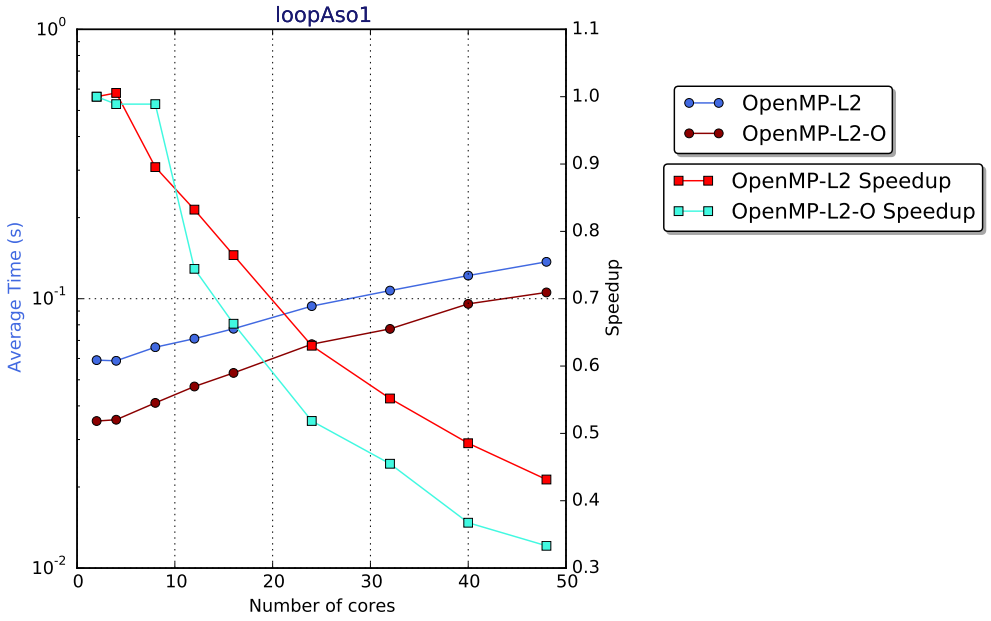


FIGURE 7.16: Performance of LoopAso1 kernel with optimization version

7.5.10 Heated Plate

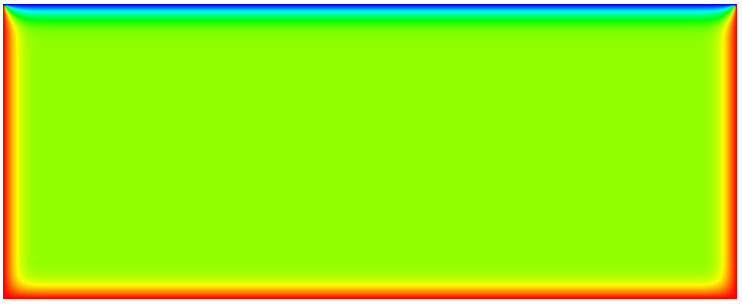


FIGURE 7.17: Heat diffusion on a 2D plate

Figure 7.17 shows the output of the heat flow in a flat metal plate with grid of 200 by 500 for heated plate simulation. This application study the 2-D steady state heat conduction in a plate using Jacobi iteration. It calculated the temperature of heat plate at each point of the interior part. To update each point, the programmer needs information about all its neighbouring points. The parallelized version using OpenMP of this simulation is described in [227]. The program code has multiple loop region that implements in parallel by using work-share construct of OpenMP. The program is classified into two parts (*Initialization* and *Computation*) based on the communication and computation intensives, each part has multiple parallel regions.

Figure 7.18 illustrates the experimental results that is performed using increase number of threads. As expected, the regular parallelism of *OpenMP-L2* implementation of Heated Plate has the best performance in the *Computation* graph. Here,

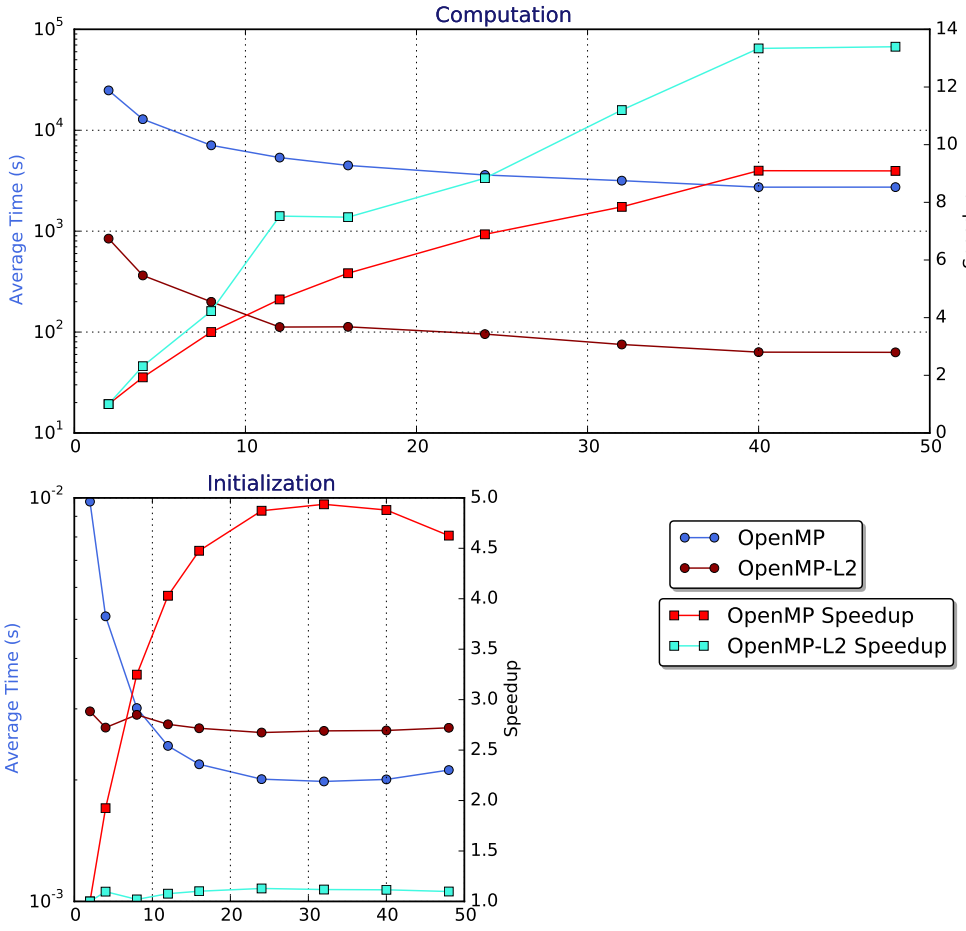


FIGURE 7.18: Performance of Heated Plate application on the SCC

Figure 7.18 shows the speedup and time performing significantly better than *OpenMP* approach, *OpenMP-L2* achieved 97.7% reduction in the execution time at 48 threads. It would make sense that this would affect a more compute-bound computation, while the *OpenMP* approach is still constrained by other resources. Of course, the programmer would reimplement this part by using *noflush* directive to gain more performance optimization. While in the *Initialization* figure, the *OpenMP* implementation has better performance when number of threads > 8. Because this part has multiple parallel region with fine-grain workload and responsible to add a reasonable initial value for the interior, therefore, increasing the number of threads consequentially added more overhead to the *OpenMP-L2* implementation.

7.6 Conclusion

This chapter introduced the OpenMP model as an optimal loop-level parallelism configuration for diverse applications. Here, the OpenMP programming model and its compiler

have been applied to a specific benchmarks and different kinds of applications. First, this chapter unveils the details of the data parallelism and new extension (*noflush*) for OpenMP compiler that used to optimize the performance. In the OpenMP API, it utilized the feature of the hardware to activate or halt the L2 cache flushing in cacheable memory mode access. To eliminate unnecessary overhead of the L2 cache traffic (hit or miss), as a result, optimizing the performance of the application. Then, it explained how this extension is incorporated into the OpenMP model. Furthermore, it implemented the reduction clause in the OpenMP programming model that used often in applications.

To evaluate the efficiency of the OpenMP design, it defined two kinds of application: bandwidth benchmarks and real applications, with diversity the parallelization scheme and the dataset size. In bandwidth benchmarks (Stream), memory performance information is gathered from detailed analysis of all memory access modes in the presence of the bandwidth and timing data. In addition, it evaluated the bandwidth of the extended version of Stream benchmarks in variety frequency settings to validate the impact of contention on OpenMP model. The results show that OpenMP implementation based on cacheable mode has the best performance.

To substantiate the correctness and potential of the OpenMP and developed extension, a large application used like SRAD, HotSpot, N-Queen, etc. This second set of experiments aims at investigating the cost of OpenMP API support to parallelization in several situations (regular and irregular) with the different accesses pattern of memory resources. For most applications, the OpenMP scales well when the OpenMP threads used the L2 cache to hide the latency, and the best performance always happens near the maximum number of hardware cores/threads. This difference in the performance owing to the class of the application and the better cache utilization on OpenMP.

The results are organized by application (Stream, SRAD, CG, LU-D, N-Queen, Mandelbrot, Helmholtz, PathFinder, HotSpot, Heated Plate, and Loop with dependency). All these applications are implemented by OpenMP parallel regions and parallel loops. Due to the alternation of communication-intensive and computation-intensive loops and finely tuning the workload through several parameters, those benchmarks provide several interesting case studies, representative of real application patterns. Usually performance hogs are found in the loop part of the program code. OpenMP helps the programmer to increase the loop performance by using loop parallelization whose iterations are distributed among the spawned threads by the parallel directive. The OpenMP compiler translates these loops into thread-based code that calls to the OpenMP runtime library to perform synchronization and scheduling. Thus, this feature is too interesting, since it possible to schedule the loop iterations over multiple threads by using only one line of code with little human intervention. The balanced parallel work achieved between

threads by allocating similar amount of work to each of them such as in static scheduling case. Here, the iteration space and communications are evenly distributed among threads, where threads reference distinct equally-sized subsets of a shared data (e.g. an array).

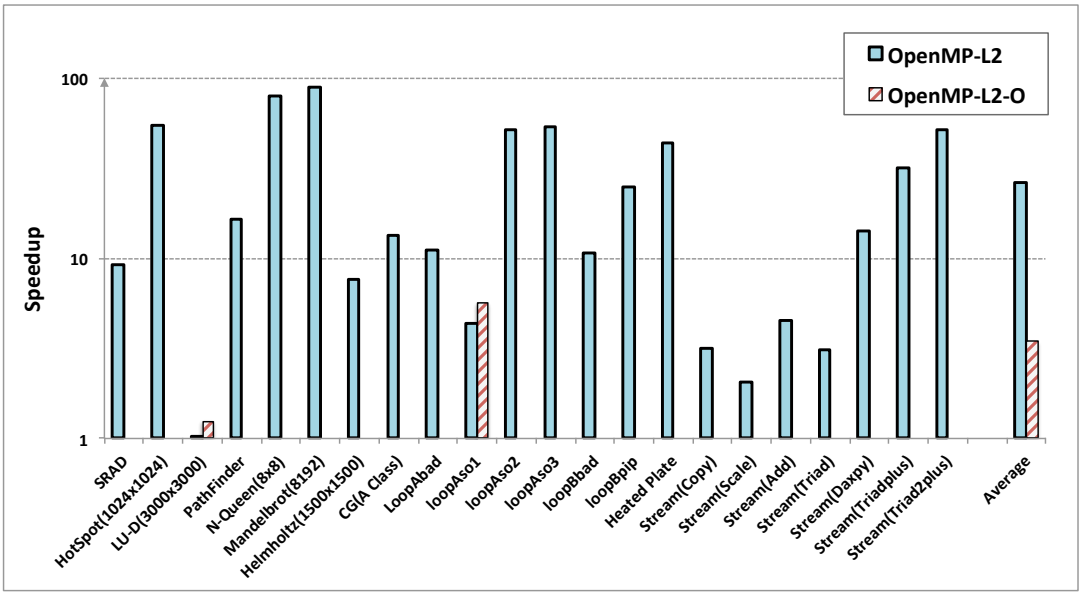


FIGURE 7.19: Speedup of several benchmarks support OpenMP-L2 against the baseline (OpenMP) for 48 cores.

Figure 7.19 shows the speedup of *OpenMP-L2/OpenMP-L2-O* against OpenMP for 48 cores only. This figure shows that on average *OpenMP-L2* allows ≈26% speedup. In some application such as N-Queen and Mandelbrot, *OpenMP-L2* achieves >90% speedup, but LU-Decomposition shows the worst scaling performance. This behavior is due to the above mentioned serialization effect of accesses on the port of the memory device that hosting the shared data. To solve this problem and achieve excellent scaling, using the cache expectantly to buffer shared data from different memory banks.

The *OpenMP-L2-O* approach that implemented by exploiting *noflush* directive in LU-Decomposition, only allowing a peak 1.2x speedup for 48 threads. This scenario happens because of the *noflush* directive enforce the runtime does not flush the cache at the end of the OpenMP parallel region. This extension allowed me to infer opportunities for optimizations that could not easily be obtained and localized. This led to transform the program resulting in both appreciably decreased coherence traffic and execution time savings such as in *loopAso1* application (5.6% reduction). In the future, I would like to assess the benefits of this directive for long-running applications on MPSoCs.

Chapter 8

Conclusions and Future Directions

Software development tasks in the scope of cluster-based many-core systems are becoming more and more complex and complicated due to the increasing number of hardware features in the design. Over the last few years, OpenMP has become a mature standard for shared memory parallel programming, which was designed more than a decade ago for SMP. Today, OpenMP has been adopted in the MPSoC domain by several researchers.

One important contribution of this dissertation is the design and implementation of a full-OpenMP programming model for non-cache coherent cluster-based many-core platforms. Concerning the programming model, OpenMP is seen as a target in a broader view due to its *productivity*. In OpenMP, parallelization can be achieved easily by inserting only the *pragma* directives at certain positions without any other efforts of controlling the threads or transferring the data explicitly. The *pragma* is a compiler directive that does not require substantial recoding and that can be ignored by a standard compiler. Practically, the effort for OpenMP programmers is quite low – even for non-expert programmers.

For this work, a new approach is chosen: Instead of relying on simulations, an entire many-core system was developed to serve as a measurement platform. The idea was to address the real system challenges by using Intel's SCC as an example for MPSoC based on cluster-on-a-chip architecture. The SCC is a researcher chip that contains a 48-core (Pentium 1) concept vehicle with no hardware cache coherence, developed for research on future many-core chips. This chip features specific innovations – such as the mesh network, the message-passing buffer or the general hardware configurability, which are common features of new multi- and many-core architectures. Moreover, future processor

architectures are likely to feature fine-grained power management functions like those provided by this chip. It is believed that the Intel SCC offers a rich set of opportunities to optimize the application behavior. As the system performance is quite poor in comparison to normal machines (i.e. desktop computers) or server systems, the performance values should not be judged on the basis of other state-of-the-art architectures.

Landing OpenMP as a shared programming model on the SCC seemed straightforward, despite several problems that made the land more difficult than expected, such as bugs in the compiler, the task of setting up the system, and the aspect of cache misbehavior. Nevertheless, it serves as a working environment on such many-core systems in order to investigate characteristics and problems which can occur in such systems, as well as possible approaches to solve them. The rest of this chapter summarizes the main contributions and conclusions of the research work described in this dissertation Section 8.1. Section 8.2 presents the potential avenues of future research opened up by this dissertation.

8.1 Contributions and Conclusions

This thesis studied how to construct an efficient OpenMP API to map threads and data parallelism onto cluster-based many-core architectures. It is attractive to support OpenMP programming on such a system, in order to increase the programmers' productivity and reduce the design/development costs in terms of time and effort for the many-core systems. As a departure from conventional techniques, this thesis is founded on working with a custom run-time library based on a modified GCC 2.6 compiler. It was discussed what issues and requirements are connected to working with the OpenMP fork-join execution model on the Intel's SCC, which was chosen as an example of cluster-based many-core system. In that sense, the significant improvements of performance can be achieved by purposefully making use of the relevant architectural features.

8.1.1 Supporting OpenMP Model on Cluster-Based Architecture

Chapter 4 described how to support OpenMP implementation on the SCC. Here, a fully automated translation and optimization system is developed to implement the OpenMP model by customizing and extending the GCC compiler. A new run-time library (**libgomp_scc**) was designed from scratch, efficiently deploying the parallelism of OpenMP in order to avoid all obstacles from using the *Pthreads* library - because Pthreads requires dedicated abstraction layers to allow threads to communicate on different cores. Furthermore, additional overhead is generated by conditional variables, by

signal handling techniques, and contact switching. *Libgomp_scc* is a low-level library-based API that is used to store all of the run-time data structures and to create and manage the OpenMP threads. This library was implemented independent of the OS and could be used in the BareMetal environment as well. In addition, it is responsible for handling and translating the shared variables.

One of the major issues in the OpenMP design is about sharing pointers of shared data between distinct threads. It is challenging to support OpenMP data sharing on SCC; making shared data from the main memory visible to all threads in presence of several OS instances, each with its own virtual memory space. To overcome this issue, a novel technique has been proposed by augmenting the GCC mechanism of marshalling the offset of the shared variable based to one common reference instead of marshalling the pointer itself within a structured construct to pass the shared object. This ultimately allows to overcome the issues related to memory aliasing when data is referenced by name. In this approach, the runtime code must ensure that the shared program data is allocated in a portion of shared memory that can be ultimately made univocally addressable by different threads. As a result, this approach has low overhead and is more scalable, meaning that the program has no limit for the number of shared objects to pass among threads.

Furthermore, a barrier algorithm has been implemented to support a realistic OpenMP programming model on a commodity many-core on-chip, considering a standard implementation specific to the SCC. To analyse the cost of the barrier synchronization and the fork/join overhead, an efficient methodology is considered to show the benefits and drawbacks of individual approaches as well as significant performance improvements for the optimal solutions.

8.1.2 Reducing the Overhead of Barrier Algorithm

Future many-core systems will feature significantly increased numbers of processing cores integrated into a chip. As a result, the barrier synchronization scheme that is required for high-level shared-memory-based programming models is becoming ever more complicated. The performance of the barrier scheme is a central aspect of the accuracy and the performance of parallel programs. Furthermore, a key to reducing run-time overheads is an efficient barrier implementation, because OpenMP relies heavily on barrier operations to control threads in parallel. Therefore, several barrier algorithms were implemented in Chapter 5, serving to support the OpenMP programming model on the SCC by considering standard implementations specific to the SCC. Using the SCC hardware primitives

and/or explicit allocation of barrier structures in the MPB serves to reduce contention and to improve performance.

The passage begins with a description of the evaluation criteria (such as Pure Overhead of barrier approaches, overhead caused by static/random load and load imbalance, NoC traffic effects, and impact of Memory Accesses) for studying the barrier synchronization with micro-benchmarks on multi-core SoC architectures. Secondly, barrier algorithms are developed based on three concepts; linear or tree-structured communication patterns, symmetric barrier phases or the signal phase may use a single shared variable as the exit signal, as well as an explicit allocation of barrier structures in the MPB and the memory access. Barrier synchronization may rely upon memory access to shared on-chip memory (e.g. MPB) or it may use configuration registers (e.g. LUT) as a simple implementation of spin-lock routines.

Based on those micro-benchmarks concerning the SCC, the experimental results highlighted a significant reduction in the overhead for barrier algorithms when using a Tree LUT-Polarity busy-wait approach. Figure 5.26 shows the speedup of all barrier algorithms against S-MSB(a) for 48 cores only. The BT-LUTB is the best barrier synchronization, allowing for a more than 88% (MO in Pure Overhead) faster synchronization than the baseline S-MSB(a) implementation. In the same context, the BT-LUTB is approximately 46.6% (MO in Pure Overhead) faster than the typical well-performing BTPB algorithm. Overall, the chapter constitutes the basis for providing an efficient and fully compliant OpenMP implementation of the MPSoC.

8.1.3 Designing Efficient Fork/Join Model

Chapter 6 introduced a hierarchy-approach to support a realistic fork/join programming model on a common many-core on-chip system by considering the standard implementation specific to the SCC. Future parallel systems will feature significantly larger numbers of processing cores integrated into a single chip. To overcome the many scalability-related bottlenecks in the many-core design, a common paradigm is a cluster-based system. Cluster-based many-cores often leverage partitioned shared memory, which is subject to the NUMA effect due to a hierarchical interconnection system. Fork/join is a widespread shared-memory programming abstraction, very appealing in the scope of developing many-core applications. However, to be able to support medium- to fine-grained parallelism which is typically encountered in embedded or HPC applications, it is necessary to lower the cost for forking and joining a large number of threads.

The goal was to optimize the fork/join runtime, using an architecture-aware, hierarchical technique for thread forking and joining, which considers the physical organization

of the platform in clusters. Architecture-agnostic sequential fork/join algorithms are not suitable for many-core systems, for two main reasons: First, placing the responsibility for recruiting a very large number of workers sequentially onto a single master thread is poorly scalable. Second, when threads are physically displaced over multiple clusters, the communication underlying fork/join support is subject to NUMA effects, increasing the cost for these primitives. In addition, these scaling issues are addressed for coordinate (spawn and join) parallel activities.

By exploiting hierarchy-approach mapping for fork and join, overheads have been reduced by up to ≈48% on the SCC by creating parallel teams of up to 48 and considering the spatial locality as well. In this work, an architecture-aware approach is presented, a hierarchical technique for thread forking and joining, which considers the physical organization of the platform in clusters.

Furthermore, several approaches are explored to improve the performance of the fork/join mechanism by efficient use of the memory hierarchy. More specifically, the runtime of OpenMP is extended to allocate the metadata close to slave threads. In architecture-agnostic algorithms (flat implementation of fork/join), this is all that can be done to reduce the latency to recruit a large number of threads or synchronize them during fork or join respectively. The experimental results of the **Mode 3** (S-L2) approach allow a 2x speedup for the maximum number of cores used.

8.1.4 Compliment and Criticism

The Stream benchmarks have been extended to evaluate the performance of parallel memory access using the OpenMP execution model. The SCC platform has one main issue: It can only have one outstanding memory operation, which, in consequence, leads to a poor memory performance. In Chapter 7, the section focusing on experimental results provides a detailed evaluation of the Stream performance achieved by different approaches. It shows the benefits and drawbacks of the individual approaches as well as the significant improvements for the optimal solutions. The proposed *OpenMP-L2* approach that utilizes the L2 cache to host the shared data temporarily is able to reach an improvement of 3218% and 5059.7% for *Traidplus* and *Traid2plus*, respectively (for 48 threads) compared to an OpenMP implementation with direct access to shared memory (off-chip).

Moreover, those benchmarks are evaluated in different frequency settings for the network and the memory to consider the influence of frequency scaling on the memory performance as is shown in Appendix B.

When mapping the application onto the hardware resources under an architecture-agnostic algorithm, this will reduce the performance due to a number of issues. High contention for the memory system (off-chip and on-chip) where shared data or metadata are allocated causes one or more threads to be deferred from access to its dataset. As a consequence, this delay with the implied barrier at the end of each parallel region leads to overall program execution distension due to the OpenMP semantics.

To address this problem, the shared data is allocated in a cached memory portion to reduce the traffic in memory access (memory controller and NoC) by exploiting the local memory (L2 cache). This approach also avoids unnecessary latency to access the data, as shown in Chapter 7. Another extensive study on the performance of several real applications is presented by considering two implementations: cached and not cached data. Experiments demonstrate that the cacheable mode implementation (*OpenMP-L2*) could lead to an average performance improvement of ≈26% in comparison to that of the uncacheable code. These implementations were experimented with in a large and complex application – and some of the applications showed significant performance gains.

However, some of the benchmark results showed a low standard deviation in cacheable mode implementation, since all experiments were done with exclusive access to the shared data with different patterns on the NUMA machine. To overcome this obstacle, a new extension for the OpenMP compiler has been proposed leading to program transformations that result in both decreased cache traffic and execution time savings. The new extension is the *noflush* directive that disables the flush routine at the end of the parallel region. As a consequence, the shared data still hosted in L2 cache and can be used again by the same thread when the next parallel region is created. Here, the programmer should use the directive with care to avoid all the issues of false sharing and the consistency model. Furthermore, the overhead of using the *reduction* clause is reduced in the applications by exploiting the one-dimensional array (i.e. hold a copy of the reduction variables) to implement it and avoid atomic access to reduction variables. Thus, it is feasible to extract high performance applications on a cluster-based processor by using the simple and easy-to-use OpenMP programming model. Moreover, a careful implementation of the shared memory abstraction upon non-uniform access is a key to achieve a significant performance improvement.

8.2 Future Work

There are abundant opportunities related to the work presented in this dissertation, and some aspects of possible future improvement based on the aforementioned ideas will be

presented in the following.

- Chapter 7 showed that the OpenMP implementation achieved a speedup of 48x using 48 cores when using some benchmarks (SRAD, HotSpot, N-Queen (c and d), and Mandelbrot(b and c)). The scalability curve shows that the OpenMP implementation based on L2 cache enabled, will entail a higher core count. Here, the scalability can be improved by redesigning and exploiting the memory hierarchy to ensure performance scalability. My proposition for future work is that data and instructions could be transferred to the local memory next to the worker core, where the memory can provide a low access latency and reduce the NoC traffic. Of course, such research requires a large scale system. Furthermore, the application source code will need to be analyzed in the future to determine the size of both data and instruction for the local memory and provide an efficient cross-node communication during the runtime.

- The future many-core architecture is still speculative, therefore, the scalability of the programming model needs to be validated on several different many-core platforms which use different cache and memory structures.

- The proposal in Chapter 7 about the extended OpenMP compiler directive to control the consistent view of shared data explicitly opens up a new field of research. The idea is to investigate the scalability and usability of this proposal in many applications, particularly by adding new analysis techniques to the compiler to estimate which data would be in the cache or not – and by using this extension implicitly to avoid error-prone. In addition, it also is possible to extend the directive to use a list of specific shared variables to stay in the cache. The other opportunity is to rebuild application algorithms in this manner to classify the shared data into *flushed* or *not flushed*. Furthermore, the performance of this extension needs to be validated on different many-core platforms and different cache levels with different flushing techniques. Also, it is a very useful extension to support the software-managed coherence design on the non-coherent cache systems.

- Exploiting the task level parallelism in OpenMP is one important area of future research. OpenMP 3.0 supports this kind of parallelism to ease expressing irregular and nested parallelism in a comparable manner [228]. This level of parallelism can be stated in different levels of granularity – such as coarse or fine granularity – which can be used in recursive functions as collections of asynchronous tasks, which is becoming more important in parallelizing compilers. In addition, it allows the programmer to express the parallelism in his application at a much finer level of detail and specify dependencies between the tasks in a good load balance fashion.

To port the tasking model in the many-core system, the researcher needs to use the memory hierarchy intensively while simultaneously minimizing parallelization overheads, since the parallelization and the memory hierarchy utilization overheads have influence on the performance scalability. Many-core systems are suitable for this kind of parallelism by exploiting the local memory to host tasks queue and rethinking of new scheduling techniques that leverage the hardware resources. My advice is to use the architectural awareness approach proposed in Chapter 6 to reduce the overhead and to design NUMA-aware parallel task constructs. Furthermore, this kind of the research will accommodate the nesting parallelism, making it more suitable for irregular codes, where loop-level parallelism creates significant load-imbalances between threads.

- To reduce the overhead associated with accessing metadata that is allocated in shared memory (off-chip or on-chip), one can use an active message mechanism and embed it in barrier primitives (in *Release* function). Thus, it is not necessary for each of the threads to gather the release flag and intensify access to the shared metadata to fetch the necessary information. Of course, the developer will avoid unnecessary latency of access to the metadata and waiting time to the memory port to respond to the request – especially, when we target an embedded many core system.

- Another direction for future work is to use predictive techniques to build an application in parallel mode that can dynamically map and schedule program threads. This could be done by determining the optimal number of threads and best scheduling policies in compile time. As a result, the performance scalability and the power consumption could be improved. Thus, future research should focus on developing strategies on how to allocate the hardware resource to the many-core candidates.

Appendix A

Intel@ Xeon Phi™ Coprocessor

This appendix depicts the Intel Xeon Phi coprocessor architecture with its features and typical programming models. It explains the micro-architectural features such as the core, the vector processing unit (VPU), the high-performance communication, fully cache coherency, and how the various units interact as the key to understand program design and optimization, such as cache organization and memory bandwidth.

A.1 Overall Architecture

The Intel coprocessor designed as many-core system based on the Intel Many Integrated Core (MIC) architecture that was known by the name *Knights Corner*. The MIC provides immense throughput as a single chip that delivers a peak performance of well above one double precision TFLOPS, to serve the needs of applications in the HPC that are used extensively of vector operations and are occasionally memory bandwidth bound. As illustrated in Figure A.1, the MIC has more than 60 x86 processor cores with long vector (SIMD) units (512-bit) connected by a on-die bidirectional ring bus [229]. Moreover, every core is a fully functional working under control of the dedicated embedded Linux μOS runs in one of the cores. Where, each core has capability of switching between up to 4 hardware threads in a round-robin fashion, providing in a total of up to 240 hardware threads available. In addition, the vector unit that allocated in every core with 64 byte registers featuring a new vector instruction set. Each core has a cache memory system that arranged in a 32KB L1 data cache, a 32KB L1 instruction cache, and a private 512KB unified L2 cache which is kept fully coherent by using a distributed tag directory system (DTDs) in the hardware. Namely, the memory system has in total 30MB of L2 cache on the die for 60-core machine. DTDs have 64 tag directors connected to the ring which are referenced after an L2 cache miss. Each tag getting an

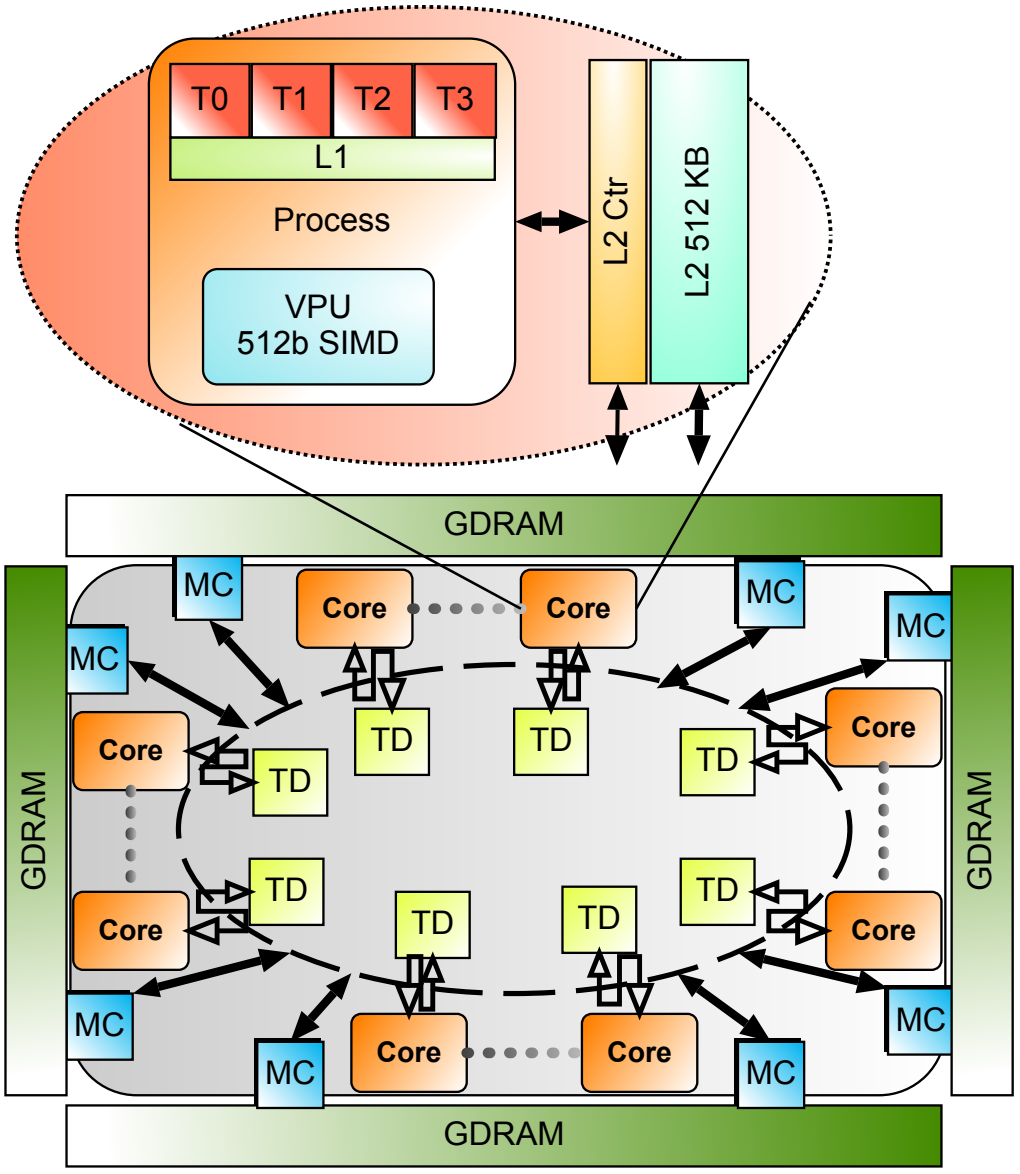


FIGURE A.1: Layout and Single Core architecture for the MIC

equal portion of the address space that is mapped to the tag directories based on hash functions. This hashing function provides a framework for more elaborate coherence protocol mechanisms than the individual cores could provide and maps each physical address to a tag directory as well.

In addition, there are 8 memory controllers providing access to 16 GDDR5 channels (8 GB of global memory) and delivering up to 5.5 GT/s with bandwidth of 352 GB/s. The Xeon Phi has a high off-chip memory access latency, although it has a high memory bandwidth. Because of the GDDR5 is used optimized for bandwidth rather than latency. Therefore, optimizing the cache/memory behavior of applications pose an obstacles for reaching the high performance because of cores are not able to hide cache/memory access

latency as out-of-order cores. XeonPhi connected to host through a special function devices such as the PCI Express system interface [230].

A.2 Programming Overview

This section provides details details on programming for MIC architecture. There are two ways an application can be execute on Xeon Phi [14]:

- Offload mode: here, the main application runs on the host and it only offloads the parallel part to the coprocessor (Xeon Phi). In this mode, the programmer could use a set of pragmas and keywords to tag code regions for execution on the coprocessor. Programmers are responsible for additional control over data transfers as well by clauses that can be added to the offload pragmas. One of the advantages of this mode is the code can contain any number of functions routines which used in any programming model such as OpenMP.

- Native mode: this mode allows the application to run independently (on the Xeon Phi only) and can communicate with host or other coprocessors [231]. Every Xeon Phi coprocessors execute a specialized Linux kernel that provids all the well-known services and interfaces to applications. In execution, we logged into the coprocessor and executed the benchmark from a standard shell. To prepare the application, the programmer needs to use *-mmic* switch with Intel Composer XE tool on the host to instruct the cross-compile to generate the application code for the Intel Xeon Phi coprocessor.

In our experiments, we will only focus on the native mode of execution, wherein an application runs exclusively on the Xeon Phi coprocessor. The native mode has some benefits such as minimal code-porting overhead from existing architectures and not having to deal with low host-to-coprocessor data transfer latency. Although we want to achieve modest performance by porting existing CPU code on Xeon Phi, that requires reasonable optimization effort to exploit its capabilities fully.

In general, to program an application on Xeon Phi, programmers need to capture both functionality and parallelism. Xeon Phi provides full capability to use many tools, programming languages, and programming models as a regular Intel Xeon processor Particularly, tools like OpenMP [204],Pthreads [62], Intel Cilk™ Plus, MPI, and OpenCL are available.

Appendix B

Stream Benchmark Results

This section contains the experimental results of Stream benchmarks in different number of threads and various frequency settings. The frequency settings for all results in the tables experiment in terms of tile, mesh and memory clock are:

- **Set0**: 533 MHz, 800 MHz and 800 MHz respectively.
- **Set1**: 800 MHz, 1600 MHz and 1066 MHz respectively.
- **Set2**: 800 MHz, 1600 MHz and 800 MHz respectively.
- **Set3**: 800 MHz, 800 MHz and 1066 MHz respectively.
- **Set4**: 800 MHz, 800 MHz and 800 MHz respectively.

B.1 Copy

B.1.1 SPMD Implementation

	Set0		**Set1**		**Set2**		**Set3**		**Set4**	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	36.2	1.110237	43.4	0.923181	42.8	0.936337	**48.0**	0.834526	41.1	0.974282
4	35.5	1.126474	44.4	0.903976	42.7	0.937560	**46.6**	0.859791	40.4	0.994285
8	33.8	1.185015	42.1	0.950129	40.5	0.988655	**44.3**	0.904105	37.8	1.060856
12	34.0	1.179863	42.5	0.944258	40.7	0.982673	**45.2**	0.888025	38.0	1.052752
16	33.3	1.206733	39.9	1.005932	39.1	1.027002	**43.7**	0.916898	36.8	1.090829
24	31.5	1.273218	33.8	1.186302	35.8	1.122327	**39.9**	1.005611	32.5	1.233831
32	31.9	1.256309	34.9	1.149346	35.0	1.148416	**40.8**	0.983402	32.3	1.242248
40	32.0	1.251268	35.1	1.142380	34.9	1.150099	**40.7**	0.985878	33.3	1.203373
48	31.4	1.277798	34.9	1.150275	33.2	1.212210	**39.1**	1.027290	33.1	1.216240

B.1.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	5.6	7.169107	**8.1**	4.919706	6.6	6.062024	7.3	5.506464	6.2	6.483650
4	11.2	3.583510	**16.2**	2.470680	13.2	3.040107	14.6	2.742333	12.4	3.234484
8	22.1	1.815184	**31.5**	1.272137	25.8	1.550915	28.8	1.388524	24.6	1.626806
12	32.3	1.242256	**44.4**	0.901489	37.0	1.081891	41.1	0.975129	35.5	1.127701
16	41.3	0.969983	**54.6**	0.732982	46.5	0.861302	51.2	0.782192	45.2	0.885842
24	55.4	0.723505	**69.6**	0.576442	61.3	0.653188	66.5	0.602433	59.5	0.674116
32	65.9	0.608272	**79.2**	0.505733	71.0	0.564642	76.5	0.523192	69.4	0.577253
40	72.8	0.550616	**84.3**	0.475176	76.8	0.521787	82.4	0.485938	75.9	0.528169
48	76.9	0.521385	**84.5**	0.474031	78.4	0.511315	84.2	0.475710	79.3	0.506126

B.1.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	5.6	7.172996	**8.1**	4.920226	6.6	6.062130	7.3	5.510737	6.2	6.483429
4	11.2	3.584343	**16.2**	2.470978	13.2	3.040172	14.6	2.742453	12.4	3.235254
8	22.1	1.814376	**31.5**	1.272529	25.8	1.550232	28.8	1.388355	24.6	1.627164
12	32.2	1.241771	**44.4**	0.901226	37.0	1.080869	41.1	0.975407	35.5	1.127627
16	41.0	0.976900	**54.6**	0.732939	46.5	0.861093	51.2	0.782522	45.2	0.885638
24	55.4	0.723355	**69.5**	0.575344	61.3	0.653664	66.5	0.602145	59.4	0.674552
32	65.9	0.607681	**79.3**	0.505410	70.9	0.565532	76.4	0.523403	69.4	0.577294
40	73.0	0.548534	**84.3**	0.474827	76.8	0.521551	82.4	0.485977	75.8	0.528161
48	77.2	0.519198	**84.5**	0.474092	78.4	0.510651	84.2	0.475629	79.2	0.506544

B.1.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	55.5	0.723808	**83.0**	0.483054	70.7	0.566644	68.7	0.583087	60.8	0.659906
4	116.0	0.346603	**165.2**	0.243158	145.2	0.276312	144.9	0.276388	129.0	0.310331
8	215.7	0.186498	**263.6**	0.151811	233.9	0.171447	260.8	0.153502	225.9	0.177149
12	242.2	0.166411	**272.8**	0.146701	245.0	0.163695	271.9	0.147215	244.2	0.164467
16	243.7	0.165152	**273.2**	0.146764	245.9	0.163212	272.8	0.147222	245.1	0.164295
24	244.9	0.166490	**273.7**	0.146552	246.4	0.162655	273.4	0.146426	246.3	0.163011
32	245.0	0.164630	**273.8**	0.146383	246.5	0.162361	273.4	0.146699	246.2	0.163371
40	244.9	0.163861	**273.8**	0.146328	246.3	0.162**Set2**	273.4	0.146671	245.8	0.163878
48	245.1	0.163880	**273.8**	0.146410	246.4	0.162764	273.4	0.146541	246.2	0.163036

B.2 Scale

B.2.1 SPMD Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	64.0	0.628853	77.1	0.520815	75.4	0.531551	**84.8**	10.472688	72.8	0.551110
4	62.6	0.640349	77.9	0.515905	75.2	0.533493	**81.8**	0.490417	71.5	0.559807
8	59.7	0.672694	74.5	0.539753	71.8	0.558702	**78.5**	0.511311	67.1	0.596861
12	60.1	0.667269	74.9	0.536301	72.1	0.556029	**80.0**	0.501858	067.5	0.594788
16	58.5	0.688161	69.9	0.573782	68.7	0.584954	**77.2**	0.519398	65.0	0.615641
24	55.2	0.725671	59.0	0.683715	62.5	0.645004	**70.1**	0.572626	57.3	0.702935
32	56.2	0.714576	61.2	0.656128	61.6	0.651128	**72.3**	0.555899	57.4	0.698507
40	56.5	0.710955	62.0	0.649780	61.6	0.651915	**72.1**	0.557064	58.9	0.682363
48	55.0	0.734284	60.4	0.664833	58.3	0.689128	**69.2**	0.581261	58.2	10.689537

B.2.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	4.9	8.112871	**7.1**	5.615253	5.8	6.892002	6.4	6.246444	5.5	7.24471
4	9.8	4.081951	**14.2**	2.817029	11.5	3.469451	12.7	3.141843	11.0	3.651125
8	19.5	2.050863	**28.0**	1.429026	22.9	1.746718	25.2	1.587849	21.8	1.836981
12	29.0	1.381158	**41.2**	0.972036	33.8	1.185424	37.3	1.073298	32.3	1.238575
16	38.1	1.050914	**53.0**	0.756273	44.2	0.906740	48.5	0.826027	42.3	0.946110
24	54.7	0.732412	**72.3**	0.554154	61.5	0.651599	67.6	0.592795	59.4	0.674580
32	68.8	0.582658	**87.8**	0.456325	76.4	0.524635	83.1	0.481677	73.3	0.546086
40	80.3	0.498869	**100.3**	0.399895	88.8	0.450817	95.7	0.418594	85.3	0.469711
48	90.4	0.443836	**107.1**	0.375315	96.6	0.415281	104.2	0.385249	95.1	0.421723

B.2.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	4.9	8.150042	**7.1**	5.615365	5.8	6.892334	6.4	6.246363	5.5	7.247151
4	9.8	4.082243	**14.2**	2.817081	11.5	3.468691	12.7	3.141929	11.0	3.652334
8	19.5	2.053203	**28.0**	1.429032	22.9	1.747683	25.2	1.588085	21.8	1.836904
12	28.9	1.384563	**41.2**	0.972215	33.8	1.184484	37.3	1.073392	32.3	1.238662
16	38.2	1.049157	**53.0**	0.756176	44.2	0.906385	48.5	0.825676	42.3	0.946102
24	54.7	0.731912	**72.3**	0.555563	61.5	0.651241	67.6	0.592662	59.4	0.673796
32	68.8	0.582819	**87.8**	0.456166	76.4	0.523631	83.1	0.481488	73.4	0.546127
40	80.4	0.498262	**100.4**	0.399345	88.9	0.451068	95.9	0.418454	85.4	0.469462
48	90.1	0.445247	**107.1**	0.374771	96.6	0.415249	104.2	0.385289	95.0	0.421686

B.2.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	57.8	0.692937	**85.3**	0.469926	72.1	0.556271	73.2	0.547245	64.6	0.620951
4	111.6	0.359324	**160.5**	0.249787	140.7	0.285075	138.5	0.289514	124.4	0.321823
8	178.7	0.224945	**207.5**	0.192986	185.6	0.215921	202.8	0.197319	183.1	0.218764
12	186.0	0.215215	**207.9**	0.192469	187.0	0.214216	207.6	0.193268	186.7	0.214632
16	186.2	0.215574	**207.9**	0.192701	187.1	0.213911	207.7	0.192994	187.0	0.214154
24	186.3	0.215344	**207.9**	0.192793	187.2	0.213924	207.7	0.192988	187.1	0.214243
32	186.3	0.216267	**207.9**	0.192826	187.1	0.214153	207.7	0.192713	187.1	0.214907
40	186.3	0.216180	**207.9**	0.192663	187.1	0.214012	207.7	0.192880	187.1	0.215146
48	186.3	0.215937	**207.9**	0.192786	187.1	0.213838	207.7	0.192833	187.0	0.215203

B.3 Add

B.3.1 SPMD Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	79.7	0.757215	96.4	0.623873	94.3	0.637941	**105.4**	0.571119	90.2	0.666662
4	77.6	0.780721	96.9	0.621172	93.2	0.645639	**101.9**	0.589429	89.0	0.676913
8	74.1	0.814188	92.3	0.652771	89.3	0.674448	**97.7**	0.616809	83.3	0.723016
12	74.6	0.807879	92.9	0.648167	89.4	0.674381	**99.4**	0.604484	83.9	0.718596
16	72.5	0.831683	86.1	0.700046	84.8	0.711164	**95.5**	0.630374	80.4	0.748903
24	68.0	0.885686	71.0	0.847241	75.7	0.796548	**85.7**	0.703430	69.7	0.867636
32	69.4	0.867920	74.3	0.809808	74.9	0.803890	**88.8**	0.677615	70.6	0.856857
40	69.8	0.864957	75.2	0.801339	75.0	0.804527	**88.6**	0.679844	72.4	0.833318
48	68.1	0.887019	73.9	0.818090	70.8	0.852064	**84.9**	0.709936	71.4	0.842973

B.3.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	4.5	13.232946	**6.5**	9.188983	5.3	11.257231	5.9	10.216860	5.0	11.902574
4	9.0	6.632311	**13.1**	4.588811	10.6	5.647385	11.7	5.108549	10.1	5.953304
8	18.0	3.328800	**25.9**	2.316316	21.1	2.854414	23.3	2.576346	20.1	2.983820
12	26.7	2.249438	**37.5**	1.600999	30.7	1.954049	34.3	1.752378	29.7	2.025852
16	34.8	1.728045	**47.8**	1.258048	39.7	1.512646	43.8	1.370380	38.3	1.567842
24	48.8	1.230439	**64.3**	0.934671	54.5	1.102231	59.9	1.002721	53.0	1.133005
32	60.3	0.996950	**76.0**	0.790521	66.4	0.904170	71.8	0.836396	64.5	0.930455
40	69.3	0.867243	**83.7**	0.717537	74.8	0.803264	80.5	0.746850	72.9	0.824259
48	75.6	0.795895	**87.0**	0.689898	78.7	0.764339	85.0	0.706424	78.9	0.762681

B.3.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	4.5	13.226623	**6.5**	9.190407	5.3	11.257639	5.9	10.216723	5.0	11.897945
4	9.1	6.619148	**13.1**	4.588870	10.6	5.648041	11.7	5.109929	10.1	5.953759
8	18.1	3.324734	**25.9**	2.316467	21.0	2.855755	23.3	2.576513	20.1	2.984029
12	26.8	2.244901	**37.5**	1.600151	30.7	1.955466	34.3	1.752072	29.7	2.025752
16	34.8	1.726776	**47.8**	1.257956	39.7	1.513756	43.8	1.370458	38.3	1.568050
24	48.9	1.228985	**64.3**	0.933517	54.5	1.101774	59.9	1.003343	53.0	1.132881
32	60.6	0.991647	**76.0**	0.789965	66.4	0.904005	71.9	0.834350	64.6	0.930481
40	69.5	0.864197	**83.8**	0.717196	74.8	0.**Set2**936	80.4	0.746587	72.9	0.823931
48	75.8	0.794204	**87.1**	0.689615	78.7	0.763291	85.0	0.706339	79.0	0.762086

B.3.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	72.4	0.829927	**110.1**	0.545961	91.1	0.660346	90.8	0.661712	80.9	0.743403
4	150.2	0.401188	**215.2**	0.280081	187.9	0.320092	185.9	0.323436	168.5	0.356261
8	273.2	0.220730	**359.4**	0.167504	315.0	0.191166	335.6	0.180499	301.6	0.199934
12	338.1	0.178216	**381.1**	0.157980	343.0	0.175332	376.4	0.159824	342.7	0.175891
16	342.6	0.176184	3**82.3**	0.157416	345.3	0.173989	379.7	0.158652	342.9	0.175638
24	343.8	0.176517	**382.2**	0.157022	345.7	0.173688	381.5	0.157406	345.5	0.175094
32	343.9	0.175999	**382.1**	0.157236	345.7	0.174274	381.8	0.157332	345.5	0.174816
40	343.8	0.175371	**382.2**	0.157066	345.5	0.174307	381.6	0.157599	345.5	0.174758
48	343.9	0.174786	**382.3**	0.157036	345.5	0.173662	381.8	0.157473	345.5	0.173922

B.4 Triad

B.4.1 SPMD Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	78.8	0.762140	95.8	0.628068	93.6	0.642641	**104.7**	0.574412	90.0	0.668895
4	76.9	0.781396	96.2	0.625377	92.5	0.650913	**101.2**	0.595519	88.2	0.681707
8	73.4	0.819988	91.6	0.657246	88.6	0.679003	**96.8**	0.620740	82.8	0.728732
12	74.0	0.813685	92.1	0.653302	88.7	0.679499	**98.6**	0.610289	83.3	0.722427
16	71.9	0.836205	85.4	0.704850	84.1	0.716366	**94.8**	0.634180	79.8	0.753446
24	67.5	0.893380	71.0	0.849662	75.0	0.802453	**85.4**	0.704421	69.6	0.865331
32	68.8	0.875226	74.2	0.812174	74.9	0.805720	**88.5**	0.679919	70.3	0.857234
40	69.2	0.873614	74.9	0.**Set3**469	74.8	0.806972	**88.3**	0.683522	72.3	0.832860
48	67.3	0.894452	73.2	0.824840	70.7	0.853580	**84.8**	0.711015	72.3	0.832860

B.4.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	4.5	13.235130	**6.5**	9.187987	5.3	11.254205	5.9	10.222946	5.0	11.887548
4	9.1	6.627948	**13.1**	4.585222	10.6	5.644911	11.8	5.107861	10.1	5.944625
8	18.0	3.325794	**25.9**	2.317385	21.1	2.851245	23.3	2.578241	20.1	2.987174
12	26.7	2.247683	**37.4**	1.604166	30.8	1.952468	34.2	1.756404	29.6	2.031015
16	34.6	1.736528	**47.6**	1.262870	39.5	1.519610	43.7	1.374658	38.1	1.577442
24	48.1	1.250272	**63.7**	0.942763	53.8	1.115817	59.4	1.012013	52.4	1.145927
32	59.0	1.017943	**74.6**	0.805188	65.4	0.919006	71.1	0.844187	63.3	0.948555
40	67.3	0.893527	**81.8**	0.734082	72.3	0.831343	79.1	0.759403	71.1	0.844794
48	72.7	0.827714	**85.5**	0.702596	76.3	0.786660	83.5	0.719199	75.6	0.795375

B.4.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	4.5	13.234046	**6.5**	9.192134	5.3	11.257530	5.9	10.223086	5.1	11.884040
4	9.1	6.630715	**13.1**	4.585493	10.6	5.645025	11.7	5.109609	10.1	5.946103
8	18.1	3.323258	**25.9**	2.317528	21.0	2.854501	23.3	2.578376	20.1	2.987720
12	26.7	2.249974	**37.4**	1.604131	30.7	1.954584	34.2	1.756384	29.6	2.031047
16	34.7	1.733641	**47.6**	1.262698	39.5	1.520056	43.7	1.374894	38.1	1.576998
24	48.4	1.241491	**63.7**	0.943499	53.8	1.116365	59.4	1.011578	52.4	1.145861
32	59.4	1.011415	**74.7**	0.805214	65.4	0.918878	70.9	0.846407	63.4	0.948420
40	67.6	0.889694	**81.9**	0.733684	72.3	0.831363	79.1	0.759285	71.1	0.843891
48	72.7	0.826499	**85.5**	0.702365	76.4	0.787058	83.6	0.718894	75.6	0.794719

B.4.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	67.9	0.885000	**103.6**	0.580108	90.7	0.663264	82.1	0.732547	76.0	0.790141
4	137.3	0.438775	**194.3**	0.309611	171.5	0.350461	167.5	0.359387	153.2	0.393014
8	217.7	0.276332	**249.4**	0.241089	225.3	0.266457	247.3	0.242679	224.1	0.268216
12	225.3	0.267076	**251.4**	0.238980	226.0	0.265904	251.1	0.239283	225.9	0.265789
16	225.3	0.266701	**251.3**	0.238906	226.3	0.265262	251.2	0.239135	226.2	0.266006
24	225.4	0.266754	**251.4**	0.238714	226.2	0.265298	251.2	0.239385	226.2	0.265519
32	225.4	0.266790	**251.3**	0.238889	226.1	0.265932	251.1	0.239471	226.1	0.265897
40	225.4	0.267435	**251.4**	0.239447	226.2	0.265789	251.2	0.239018	226.2	0.267420
48	225.3	0.266792	**251.3**	0.238898	226.2	0.266239	251.2	0.239686	226.2	0.265411

B.5 Daxpy

B.5.1 SPMD Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	78.5	0.510463	0100.7	0.398827	97.2	0.412123	**106.9**	0.375416	96.0	0.417417
4	77.0	0.520794	99.7	0.403170	97.0	0.412991	**104.1**	0.385063	92.8	0.431829
8	75.1	0.535685	98.4	0.408134	94.7	0.423785	**102.0**	0.393914	90.1	0.444889
12	75.9	0.529089	99.3	0.404238	95.5	0.419500	**103.7**	0.387096	91.1	0.440446
16	75.3	0.533681	96.3	0.416876	94.4	0.425250	**102.4**	0.391396	89.9	0.447609
24	73.9	0.544583	90.6	0.444272	91.0	0.442077	**98.8**	0.405840	85.3	0.470495
32	75.4	0.533550	93.5	0.431897	92.6	0.433518	**101.6**	0.396065	87.8	0.456693
40	76.7	0.523902	95.0	0.421805	94.1	0.426986	**102.8**	0.390580	89.9	0.447209
48	76.6	0.527543	95.3	0.421682	93.2	0.430503	**102.5**	0.393109	90.0	0.447190

B.5.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	3.0	13.327026	**4.4**	9.188553	3.6	11.260496	3.9	10.255205	3.3	11.944341
4	6.0	6.619742	**8.7**	4.583197	7.1	5.645896	7.8	5.106695	6.7	5.932241
8	12.0	3.329902	**17.1**	2.341643	14.0	2.867238	15.5	2.579233	13.4	2.991449
12	17.6	2.277227	**24.4**	1.637988	20.1	1.985748	22.3	1.792714	19.4	2.061593
16	22.6	1.771910	**30.6**	1.306876	25.6	1.562727	28.3	1.413543	24.8	1.617344
24	31.0	1.290976	**39.7**	1.007748	34.2	1.171568	37.4	1.069916	33.4	1.198982
32	37.0	1.082538	**45.6**	0.87**Set2**3	40.3	0.992758	43.8	0.914393	39.3	1.017733
40	41.1	0.974455	**47.7**	0.839185	43.8	0.914398	47.5	0.843793	43.3	0.923488
48	43.5	0.919607	**47.7**	0.839169	45.1	0.888172	47.7	0.839543	44.9	0.891037

B.5.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	3.0	13.330473	**4.4**	9.190627	3.6	11.264521	3.9	10.257759	3.3	11.944236
4	6.0	6.616026	**8.7**	4.583327	7.1	5.650144	7.8	5.105230	6.7	5.935132
8	12.0	3.326990	**17.1**	2.341998	13.9	2.870530	15.5	2.579327	13.4	2.991545
12	17.6	2.279342	**24.4**	1.638980	20.2	1.985796	22.3	1.792548	19.4	2.062167
16	22.6	1.771013	**30.6**	1.306762	25.6	1.565255	28.3	1.413561	24.8	1.617518
24	30.9	1.295008	**39.8**	1.007247	34.2	1.172095	37.4	1.069378	33.4	1.198145
32	37.2	1.077223	**45.6**	0.877640	40.3	0.993162	43.8	0.914420	39.4	1.017386
40	41.3	0.970374	**47.7**	0.839165	43.8	0.914377	47.5	0.843322	43.4	0.923663
48	43.6	0.919640	**47.7**	0.839242	45.1	0.888157	47.7	0.839204	45.0	0.890590

B.5.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	78.1	0.514267	**116.7**	0.343221	107.3	0.373966	106.0	0.378858	100.8	0.398711
4	159.2	0.252584	**234.6**	0.170761	215.4	0.186344	215.8	0.186120	203.3	0.199143
8	318.3	0.127168	**466.3**	0.087489	430.9	0.093134	436.3	0.091918	407.6	0.099791
12	465.1	0.086209	**646.9**	0.062095	597.6	0.067364	620.5	0.064603	576.2	0.070542
16	571.9	0.070176	**682.1**	0.058917	634.1	0.063141	679.7	0.059242	627.9	0.064587
24	629.4	0.063623	**700.0**	0.057185	649.5	0.061639	697.7	0.057686	647.5	0.062040
32	644.0	0.062923	**700.7**	0.057201	649.9	0.061655	698.6	0.057834	648.8	0.062111
40	644.6	0.063807	**700.5**	0.057855	650.6	0.061633	698.1	0.057491	649.2	0.062662
48	645.1	0.064729	**699.7**	0.057531	648.0	0.062618	698.5	0.057817	649.2	0.064532

B.6 Triadplus

B.6.1 SPMD Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	77.3	0.777156	100.6	0.597552	98.3	0.611996	**106.5**	0.564395	96.1	0.626417
4	75.9	0.798558	99.5	0.603480	97.2	0.618882	**104.1**	0.576921	94.3	0.638932
8	73.7	0.815835	97.1	0.620133	94.8	0.636108	**101.3**	0.593547	90.4	0.664705
12	74.1	0.810894	97.4	0.617631	94.8	0.635653	**102.4**	0.587257	90.9	0.662613
16	73.1	0.823218	94.9	0.634532	93.0	0.646868	**100.6**	0.598238	89.1	0.676757
24	71.2	0.845754	89.9	0.669494	89.4	0.674162	**96.5**	0.622623	84.9	0.708222
32	71.5	0.841369	90.6	0.664829	89.5	0.672034	**97.4**	0.618381	85.5	0.702746
40	71.5	0.842547	90.3	0.667955	89.1	0.676958	**97.1**	0.621071	85.9	0.700992
48	70.6	0.853655	89.1	0.674796	87.4	0.689903	**95.6**	0.629882	84.5	0.713224

B.6.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	1.8	34.043752	**2.6**	23.473781	2.1	29.004668	2.3	26.209680	2.0	30.478332
4	3.5	16.969317	**5.1**	11.713989	4.1	14.482926	4.6	13.096740	3.9	15.194248
8	7.1	8.507367	**10.1**	5.935371	8.2	7.318432	9.1	6.609601	7.9	7.626622
12	10.4	5.775436	**14.6**	4.103837	12.0	5.007043	13.3	4.528223	11.5	5.206191
16	13.5	4.459666	**18.5**	3.238162	15.4	3.894544	17.0	3.532297	14.9	4.031286
24	18.9	3.183200	**24.7**	2.425696	21.0	2.856676	23.1	2.602101	20.5	2.929970
32	23.2	2.590802	**29.0**	2.068947	25.4	2.359834	27.6	2.178334	24.8	2.420412
40	26.5	2.269101	**31.7**	1.894211	28.4	2.114156	30.6	1.962541	27.8	2.160227
48	28.3	2.126788	**32.7**	1.838182	29.6	2.025863	32.0	1.877023	29.4	2.044024

B.6.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	1.8	34.056308	**2.6**	23.477022	2.1	29.000588	2.3	26.211882	2.0	30.466803
4	3.5	16.975184	**5.1**	11.714244	4.1	14.481058	4.6	13.101340	3.9	15.201879
8	7.0	8.516113	**10.1**	5.935307	8.2	7.317821	9.1	6.609382	7.9	7.627872
12	10.4	5.777698	**14.6**	4.103393	12.0	5.008415	13.3	4.528164	11.5	5.205164
16	13.5	4.460657	**18.5**	3.237388	15.4	3.895642	17.0	3.533502	14.9	4.030668
24	18.9	3.179680	**24.7**	2.426254	21.0	2.856805	23.1	2.602040	20.5	2.929360
32	23.2	2.586839	**29.0**	2.068783	25.4	2.360264	27.6	2.178507	24.8	2.419449
40	26.5	2.269700	**31.7**	1.893454	28.4	2.114005	30.6	1.961993	27.8	2.157894
48	28.2	2.127939	**32.7**	1.837713	29.7	2.025766	32.0	1.876151	29.4	2.042643

B.6.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	57.3	1.050260	**86.5**	0.694925	80.9	0.743282	81.8	0.734588	78.5	0.765776
4	114.7	0.524753	**173.5**	0.346867	161.9	0.371347	164.1	0.366321	157.5	0.382737
8	341.8	0.175700	**347.2**	0.173077	324.7	0.185017	328.1	0.182921	316.0	0.190073
12	452.5	0.132706	**514.5**	0.116809	481.9	0.125273	488.7	0.122935	469.9	0.127800
16	667.4	0.090152	**678.3**	0.089012	632.1	0.094989	646.5	0.092957	620.4	0.097256
24	667.4	0.090152	**974.6**	0.061611	907.2	0.066581	941.4	0.063865	889.2	0.068438
32	873.1	0.070740	**1024.4**	0.058656	951.6	0.063149	1016.5	0.060712	946.1	0.064511
40	926.3	0.066619	**1038.7**	0.059361	964.6	0.062366	1034.4	0.058454	962.2	0.064382
48	937.5	0.067479	**1045.3**	0.059218	967.5	0.064958	1041.4	0.058955	965.1	0.066455

B.7 Triad2plus

B.7.1 SPMD Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	48.2	1.246583	65.7	0.915890	64.8	0.929204	**68.4**	0.879151	64.0	0.939473
4	47.6	1.260955	65.5	0.918034	64.5	0.934168	**67.4**	0.893090	63.3	0.950228
8	46.9	1.283335	64.8	0.928269	63.7	0.944308	**66.5**	0.904370	61.8	0.975329
12	47.1	1.277587	65.0	0.925289	63.9	0.941895	**67.0**	0.898966	62.1	0.967968
16	46.9	1.282735	64.4	0.934097	63.5	0.947052	**66.6**	0.905934	61.7	0.975833
24	46.3	1.299428	63.3	0.948570	62.7	0.960781	**65.5**	0.918910	60.4	0.997662
32	46.5	1.296311	63.4	0.949067	62.8	0.959253	**65.8**	0.914645	60.7	0.992232
40	46.6	1.293106	63.4	0.949966	62.8	0.957577	**65.8**	0.914283	61.0	0.987786
48	46.3	1.299847	63.1	0.953272	62.4	0.964618	**65.4**	0.920161	60.6	0.993618

B.7.2 OpenMP Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	0.6	105.903492	**0.8**	73.356179	0.7	90.228898	0.7	81.349673	0.6	94.609974
4	1.1	53.027781	**1.6**	36.688739	1.3	45.257900	1.5	40.752101	1.3	47.320804
8	2.3	26.552218	**3.2**	18.528727	2.6	22.715467	2.9	20.496776	2.5	23.738731
12	3.4	17.839710	**4.8**	12.620123	3.9	15.364619	4.3	13.880404	3.7	16.043412
16	4.4	13.603289	**6.1**	9.799190	5.1	11.799745	5.6	10.696685	4.9	12.289935
24	6.3	9.500186	**8.5**	7.101675	7.2	8.389655	7.8	7.649669	6.9	8.680283
32	7.9	7.561728	**10.3**	5.835549	8.9	6.750112	9.7	6.210379	8.6	6.969449
40	9.3	6.434221	**11.6**	5.164972	10.2	5.867442	11.0	5.432676	10.0	6.027160
48	10.4	5.800034	**12.3**	4.868205	11.0	5.437250	11.9	5.038953	11.0	5.4**Set3**28

B.7.3 OpenMP_O Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	0.6	105.836510	**0.8**	73.377993	0.7	90.252416	0.7	81.370073	0.6	94.596363
4	1.1	52.956525	**1.6**	36.696213	1.3	45.266656	1.5	40.758725	1.3	47.358667
8	2.3	26.521742	**3.2**	18.534344	2.6	22.717782	2.9	20.502107	2.5	23.750263
12	3.4	17.822047	**4.8**	12.620852	3.9	15.365757	4.3	13.882999	3.7	16.049227
16	4.4	13.584448	**6.1**	9.799956	5.1	11.800398	5.6	10.702506	4.9	12.295637
24	6.3	9.491250	**8.4**	7.103745	7.2	8.390989	7.8	7.651425	6.9	8.681139
32	8.0	7.543438	**10.3**	5.835968	8.9	6.750422	9.7	6.211053	8.6	6.971946
40	9.3	6.435855	**11.6**	5.166008	10.2	5.867692	11.0	5.433062	10.0	6.025467
48	10.3	5.803137	**12.3**	4.868988	11.0	5.437473	11.9	5.038363	11.0	5.479164

B.7.4 OpenMP_L2 Implementation

	Set0		Set1		Set2		Set3		Set4	
#	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec	MB/s	Sec
2	23.0	2.611984	**34.7**	1.730257	33.7	1.782386	33.7	1.783209	33.4	1.800523
4	46.0	1.307121	**69.4**	0.866659	67.3	0.892631	67.1	0.895883	66.7	0.900482
8	92.0	0.654370	**138.6**	0.433328	134.3	0.448240	133.9	0.448345	133.4	0.450792
12	137.8	0.435884	**207.3**	0.290576	200.9	0.298912	200.3	0.300155	199.5	0.301485
16	183.3	0.328338	**275.3**	0.218111	266.9	0.225884	266.5	0.225745	265.2	0.227516
24	273.2	0.220770	**409.7**	0.146491	397.7	0.150991	397.3	0.151277	395.0	0.152716
32	362.0	0.166051	**540.4**	0.111194	526.2	0.114152	527.5	0.114366	520.1	0.115517
40	448.9	0.133831	**664.7**	0.090378	649.9	0.092427	654.8	0.091802	639.6	0.094311
48	534.2	0.112430	**784.9**	0.076581	758.2	0.079439	776.3	0.078120	750.7	0.080010

Appendix C

OpenMP History

This appendix shows the nice graphic that published by [232] to depict the history of the OpenMP specifications.

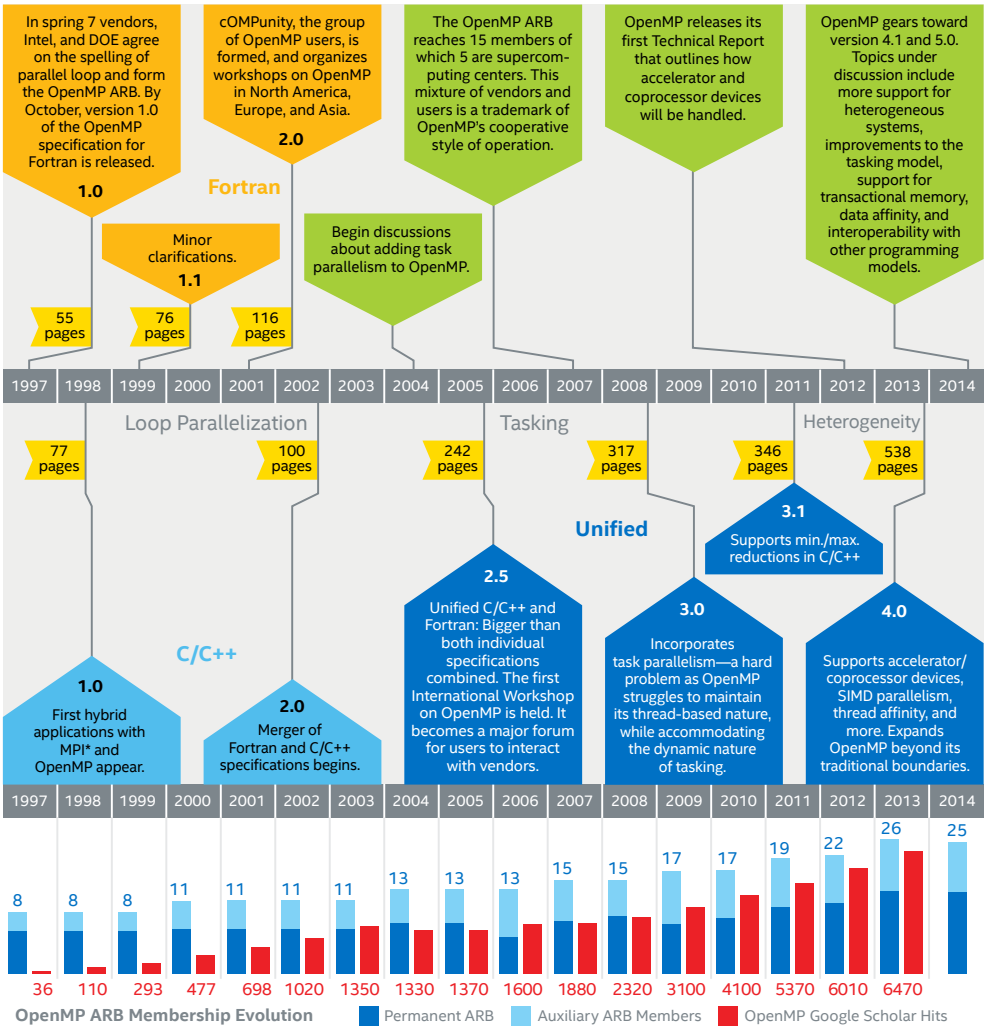


FIGURE C.1: The History of OpenMP

Bibliography

[1] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[2] ITRS. International technology roadmap for semiconductors-system drivers. 2011. URL http://www.itrs.net/links/2011itrs/2011chapters/2011sysdrivers.pdf.

[3] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope. Efficient memory copy operations on the 48-core intel scc processor. In *MARC Symposium*, pages 13–18, 2011.

[4] Intel Corporation. *SCC External Architecture Specification (EAS) - Revision 1.1*, November 2010. URL http://communities.intel.com/docs/DOC-5852.

[5] C. Rosales. Porting to the intel xeon phi: Opportunities and challenges. 2013.

[6] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[7] R. R. Schaller. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[8] N. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, December 2003. ISSN 0018-9162.

[9] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2007.

[10] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[11] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[12] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, September 2007.

[14] J. Jeffers and J. Reinders. *Intel® Xeon Phi coprocessor high-performance programming*. Elsevier Waltham (Mass.), Amsterdam, Boston (Mass.), Heidelberg..., et al., 2013. ISBN 978-0-12-410414-3.

[15] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *J. Syst. Softw.*, 81(11): 1920–1930, November 2008.

[16] OpenMP Architecture Review Board. Openmp c and c++ application program interface -version 1.0, October 1998. URL http://www.openmp.org.

[17] Gomp. An openmp implementation for gcc. URL http://www.openmp.org/mp-documents/spec30.pdf.

[18] J. Held, J. Bautista, and S. Koehl. White paper from a few cores to many: A tera-scale computing research review, 2006.

[19] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[20] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[21] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005. ISSN 0018-8646.

[22] C. Pinto, S. Raghav, A. Marongiu, M. Ruggiero, D. Atienza, and L. Benini. Gpgpu-accelerated parallel and fast simulation of thousand-core platforms. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and*

Grid Computing, CCGRID '11, pages 53–62, Washington, DC, USA, 2011. IEEE Computer Society.

[23] Kalray. Mppa manycore: A multicore processors family. URL http://www.kalray.eu/products/mppa-manycore-a-multicore-processors-family-13/.

[24] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM.

[25] Adapteva. Epiphany architecture reference. URL http://adapteva.com/docs/epiphany_arch_ref.pdf.

[26] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98–589, Feb 2007.

[27] Plurality. The hypercore processor, 2010. URL http://www.plurality.com/hypercore.html.

[28] D. H. Woo and H. S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[29] W. mei W. Hwu, S. Ryoo, S. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC*, pages 754–759. IEEE, 2007.

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[31] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] A. C. Sodan. Message-passing and shared-data programming models - wish vs. reality. In *HPCS*, pages 131–148. IEEE Computer Society, 2005.

[33] M.J. Sottile, T.G. Mattson, and C.E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2011.

[34] OpenMP Architecture Review Board. Openmp application program interface v.3.0. 2008. URL http://www.openmp.org/mp-documents/spec30.pdf.

[35] T. G. Mattson, B. A. Sanders, and B. Massingill. *Patterns for parallel programming*. 2005.

[36] A. Geist, A. G. Ornl, W. S. Nas, and T. Skjellum. Mpi-2: Extending the message-passing interface, 1996.

[37] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[38] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7): 33–38, July 2008.

[39] L. Geng, D. Zhang, M. Gao, Y. Chen, and G. Du. Prototype design of cluster-based homogeneous multiprocessor system-on-chip. In *Proceedings of the 3rd International Conference on Anti-Counterfeiting, Security, and Identification in Communication*, ASID'09, pages 311–315, Piscataway, NJ, USA, 2009. IEEE Press.

[40] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mpsoc) technology. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(10):1701–1713, October 2008. ISSN 0278-0070.

[41] W. Jeun and S. Ha. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 44–49, Washington, DC, USA, 2007. IEEE Computer Society.

[42] F. Liu and V. Chaudhary. Extending openmp for heterogeneous chip multiprocessors. In *32nd International Conference on Parallel Processing (ICPP 2003), 6-9 October 2003, Kaohsiung, Taiwan*, page 161. IEEE Computer Society, 2003.

[43] F. Liu and V. Chaudhary. A practical openmp compiler for system on chips. In *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, pages 54–68, Berlin, Heidelberg, 2003. Springer-Verlag.

[44] K. O'Brien, K. M. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[45] A. Marongiu, P. Burgio, and L. Benini. Supporting openmp on a multi-cluster embedded mpsoc. *Microprocess. Microsyst.*, 35(8):668–682, November 2011.

[46] S. L Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[47] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, and C. A. Nelson. Extending openmp for numa machines. In *Supercomputing, ACM/IEEE 2000 Conference*, page 48. ACM, Nov 2000.

[48] A. J. Dorta, J. M. Badía, E. S. Quintana-Ortí, and F. de Sande. Implementing openmp for clusters on top of mpi. In *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 148–155. Springer, 2005.

[49] A. P. Rendell, B. M. Chapman, and M. S. Müller, editors. *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, 2013. Springer.

[50] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguadé, and D. Jiménez-González. Openmp extensions for fpga accelerators. In *ICSAMOS*, pages 17–24. IEEE, 2009.

[51] A. Basumallik and R. Eigenmann. Towards automatic translation of openmp to mpi. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 189–198, New York, NY, USA, 2005. ACM.

[52] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 119–128, New York, NY, USA, 2006. ACM.

[53] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. From openmp to mpi: first experiments of the step source-to-source transformation tool. In *PARCO*, pages 669–676, 2009.

[54] J. Wang, C. Hu, J. Zhang, and J. Li. Openmp compiler for distributed memory architectures. *SCIENCE CHINA Information Sciences*, 53(5):932–944, 2010.

[55] A. J. Dorta, P. López, and F. de Sande. Basic skeletons in llc. *Parallel Computing*, 32(7-8):491–506, 2006.

[56] L. Huang, B. Chapman, and R. Kendall. Openmp for clusters. In *the Fifth European Workshop on OpenMP, EWOMP 2003*, pages 22–26, 2003.

[57] L. Huang, B. M. Chapman, and Z. Liu. Towards a more efficient implementation of openmp for clusters via translation to global arrays. *Parallel Computing*, 31 (10-12):1114–1139, 2005.

[58] D. Eachempati, L. Huang, and B. M. Chapman. Strategies and implementation for translating openmp code for clusters. In *HPCC*, pages 420–431, 2007.

[59] J. Merrill. GENERIC and GIMPLE: a new tree representation for entire functions. In *GCC developers summit 2003*, pages 171–180, 2003.

[60] D. Novillo. Openmp and automatic parallelization in gcc. In *the Proceedings of the GCC Developers*, 2006.

[61] A. Kouadri-Mostefaoui, D. Millot, C. Parrot, and F. Silber-Chaussumier. Prototyping the automatic generation of mpi code from openmp programs in gcc. In *First International Workshop on GCC Research Opportunities held in conjunction with the 4th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, january 2009.

[62] F. Mueller. Pthreads library interface. Technical report, 1999.

[63] M. A. Bauer. High performance computing: The software challenges. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 11–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-741-4.

[64] H. Iwashita, M. Kaneko, M. Aoki, K. Hotta, and G. M. van Waveren. On the implementation of openmp 2.0 extensions in the fujitsu primepower compiler. In Alexander V. Veidenbaum, Kazuki Joe, Hideharu Amano, and Hideo Aiso, editors, *ISHPC*, volume 2858 of *Lecture Notes in Computer Science*, pages 523–528. Springer, 2003.

[65] Sun Microsystems. Sun studio 12: Openmp api user's guide, 2007. URL docs.oracle.com/cd/E19205-01/820-7883/820-7883.pdf.

[66] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen. Compiler and runtime support for running openmp programs on pentium-and itanium-architectures. In *HIPS*, pages 47–55, 2003.

[67] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P. Wacrenier. An efficient openmp runtime system for hierarchical architectures. In Barbara M. Chapman, Weimin Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *IWOMP*, volume 4935 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2007.

[68] C. Brunschen and M. Brorsson. Odinmp/ccp - a portable implementation of openmp for c. *Concurrency*, 12(12):1193–1203, 2000. QC 20100525.

[69] V. Dimakopoulos, E. Leontiadis, and G. Tzoumas. A portable c compiler for openmp v.2.0. In *EWOMP 2003, 5th European Workshop on OpenMP, Aachen, Germany*, pages 5–1, September 2004.

[70] C. Liao, O. Hern, B. Chapman, W. Chen, and W. Zheng. Openuh: An optimizing, portable openmp compiler. In *12th Workshop on Compilers for Parallel Computers*, 2006.

[71] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *European Workshop on OpenMP (EWOMP'04). Pp*, pages 103–109, 2004.

[72] K. S. Gatlin and P. Isensee. Openmp and c++: Reap the benefits of multithreading without all the work. In *MSDN Magazine*, October 2005. URL http://msdn.microsoft.com/en-us/magazine/cc163717.aspx.

[73] QLogic. Pathscale compiler suite user guide, v.3.0, 2007.

[74] E. Ayguadé, M. González, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. Nanoscompiler: A research platform for openmp extensions. In *the First EuropeanWorkshop on OpenMP*, pages 27–31, 1999.

[75] S. Karlsson. Odinmp homepage, August 2004. URL http://www.odinmp.com.

[76] M. Sato, M. S. Shigehisa, K. Kusano, and Y. Tanaka. Design of openmp compiler for an smp cluster. In *EWOMP '99*, pages 32–39, 1999.

[77] J. Chow, L. E. Lyon, and V. Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *CASCON*, page 5. IBM, 1996.

[78] V. V. Dimakopoulos and A. Georgopoulos. The ompi openmp/c compiler. In *PCI2005, 10th Panhellenic Conference on Informatics*, pages 153–162, November 2005.

[79] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[80] J. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed, 1995.

[81] A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *ISCA*, pages 106–117. IEEE Computer Society, 1994.

[82] R. Pinto, R. Bianchini, and C. L. de Amorim. Comparing latency-tolerance techniques for software dsm systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(11): 1180–1190, 2003.

[83] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. ISSN 0018-9340.

[84] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990. ISSN 0163-5964.

[85] M. Sato, H. Harada, and Y. Ishikawa. Openmp compiler for a software distributed shared memory system scash (extended abstract). In *In WOMPAT 2000*, 2000.

[86] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa. Cluster-enabled openmp: An openmp compiler for the scash software distributed shared memory system. *Sci. Program.*, 9(2,3):123–130, August 2001. ISSN 1058-9244.

[87] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory. In *HPC Asia 2000*, May 2000.

[88] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[89] J. Hoeflinger. Extending openmp to clusters, 2006.

[90] C. Terboven, D. an Mey, D. Schmidl, and M. Wagner. First experiences with intel cluster openmp. In *IWOMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2008.

[91] 16th international parallel and distributed processing symposium (ipdps 2002), 15-19 april 2002, fort lauderdale, fl, usa, cd-rom/abstracts proceedings. In *IPDPS*, pages 171–178. IEEE Computer Society, 2002. ISBN 0-7695-1573-8.

[92] W. Jeun, Y. Kee, and S. Ha. Improving performance of openmp for smp clusters through overlapped page migrations. In M. S. Müller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, editors, *IWOMP*, volume 4315 of *Lecture Notes in Computer Science*, pages 242–252. Springer, 2006.

[93] Y. Kee, J. Kim, and S. Ha. Parade: An openmp programming environment for smp cluster systems. In *SC*, page 6. ACM, 2003.

[94] S. Min and R. Eigenmann. Optimizing irregular shared-memory applications for clusters. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 256–265, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3.

[95] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff. A hybrid approach of openmp for clusters. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 75–84, New York, NY, USA, 2012. ACM.

[96] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. Upc language specification v1.1.1, October 2003.

[97] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13): 825–836, 1998. ISSN 1096-9128.

[98] C. Inc. Chapel language specification 0.785.

[99] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, K. Kielstra, A.and Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[100] ScaleMP. The versatile smp (vsmp) architecture and solutions based on vsmp foundation, white paper, 2009.

[101] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user"s guide to pvm parallel virtual machine. Technical report, Knoxville, TN, USA, 1991.

[102] VMware Inc. Performance of vmware vmi, technical paper, 2008.

[103] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[104] A. Marongiu, P. Burgio, and L. Benini. Supporting openmp on a multi-cluster embedded mpsoc. *Microprocess. Microsyst.*, 35(8):668–682, November 2011.

[105] B. M. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *IPDPS*, pages 1–8. IEEE, 2009.

[106] J. del Cuvillo, W. Zhu, and G. R. Gao. Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip. In *Conf. Computing Frontiers*, pages 41–50. ACM, 2006.

[107] Yoshihiko Hotta, Mitsuhisa Sato, Yoshihiro Nakajima, and Yoshinori Ojima. Openmp implementation and performance on embedded renesas m32r chip multiprocessor. In *Proceedings of the European Workshop on OpenMP (EWOMP)*, 2004.

[108] Y. Kim, S. Kwon, W. Jeun, S. Ha, and Y. Paek. An openmp translator with retargetable parallel programming model for mpsoc, 2007.

[109] A. Marongiu and L. Benini. Efficient openmp support and extensions for mpsocs with explicitly managed memory hierarchy. In *DATE*, pages 809–814. IEEE, 2009.

[110] A. Marongiu and L. Benini. An openmp compiler for efficient use of distributed scratchpad memory in mpsocs. *IEEE Trans. Computers*, 61(2):222–236, 2012.

[111] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis. Deploying openmp on an embedded multicore accelerator. In *ICSAMOS*, pages 180–187. IEEE, 2013.

[112] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini. Enabling fine-grained openmp tasking on tightly-coupled shared memory clusters. In *DATE*, pages 1504–1509, 2013.

[113] J. Lee, M. Sato, and T. Boku. Openmpd: A directive-based data parallel language extension for distributed memory systems. In *Parallel Processing - Workshops, 2008. ICPP-W '08. International Conference on*, pages 121–128, Sept 2008.

[114] J. Lee and M. Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 413–420, Sept 2010.

[115] T. Nomizu, D. Takahashi, Jinpil Lee, T. Boku, and M. Sato. Implementation of xcalablemp device acceleration extention with opencl. In *Parallel and Distributed*

Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 2394–2403, May 2012.

[116] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: The programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[117] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. Scc: A flexible architecture for many-core platform research. *Computing in Science Engineering*, 13(6):79–83, Nov 2011.

[118] Intel Corporation. Shared memory on rock creek, 2010. URL https://communities.intel.com/servlet/JiveServlet/previewBody/5644-102-1-8755/HowtoHijackMemory.pdf.

[119] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha. A case for software managed coherence in many-core processors. In *Proceedings of 2nd Usenix Workshop on Hot Topics in Parallelism, Usenix Assoc.*, 2010.

[120] R. Rotta. On efficient message passing on the intel scc. In *MARC Symposium*, pages 53–58, 2011.

[121] R. Bakker. Exploring the intel single-chip cloud computer and its possibilities for svp, 2011.

[122] Intel Corporation. *How to Flush the L2 Cache*, February 2011. URL https://communities.intel.com/docs/DOC-6144.

[123] Intel Corporation. *The SccKit 1.4.x User's Guide - Revision 1.1*, 2011. URL https://communities.intel.com/docs/DOC-6241.

[124] J. Sobania, P. Tröger, and A. Polze. Linux operating system support for the scc platform - an analysis. In *MARC Symposium*, pages 31–34. KIT Scientific Publishing, Karlsruhe, 2011.

[125] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe. Early experience with the barrelfish os and the single-chip cloud computer. In *MARC Symposium*, pages 35–39. KIT Scientific Publishing, Karlsruhe, 2011.

[126] Intel Corporation. *SCC Programmer's Guide - Revision 1.0*, 2012. URL https://communities.intel.com/docs/DOC-5684.

[127] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. R. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmer's view. In *SC*, pages 1–11, 2010.

[128] R. F. Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1): 73–83, February 2011.

[129] Microsoft Research. Visual studio add-in and bare-metal environment for intel scc, Mar 2011. URL http://research.microsoft.com/en-us/downloads/37ccb116-c67d-4c44-9181-898889b8352d/.

[130] Intel MARC forums. Eti's scc development framework, Aug 2011. URL http://communities.intel.com/thread/17643/.

[131] M. Ziwisky and D. Brylow. Baremichael: A minimalistic bare-metal framework for the intel scc. In *MARC Symposium*, pages 66–71. ONERA, The French Aerospace Lab, 2012.

[132] S. Lankes, P. Reble, C. Clauss, and O. Sinnen. The Path to MetalSVM: Shared Virtual Memory for the SCC. In *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*, Potsdam, Germany, December 2011. Belonged to the Winner of the Best Paper Award - 2nd Place.

[133] S. Lankes, P. Reble, O. Sinnen, and C. Clauss. Revisiting shared virtual memory systems for non-coherent memory-coupled cores. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 45–54, New York, NY, USA, 2012. ACM.

[134] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[135] T. Rauber and G. Rünger. *Parallel Programming - for Multicore and Cluster Systems*. Springer, New York, 2010.

[136] I. A. Comprés Ureña, M. Riepen, and M. Konow. Rckmpi - lightweight mpi implementation for intel's single-chip cloud computer (scc). In *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2011.

[137] S. Christgau, S. Kiertscher, and B. Schnor. The benefit of topology awareness of mpi applications on the scc. In *MARC Symposium*, pages 47–51. KIT Scientific Publishing, Karlsruhe, 2011.

[138] I. A. Comprés Ureña and M. Riepen. Improved rckmpi's sccmpb channel: Scaling and dynamic processes support. In *of the 4th Many-core Applications Research Community (MARC) Symposium*, Lecture Notes in Computer Science, pages 1–7. P. Troger and A. Polze, Eds., no. 55, Potsdam.

[139] S. Christgau and B. Schnor. Awareness of mpi virtual process topologies on the single-chip cloud computer. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 529–536, May 2012.

[140] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the intel scc many-core processor. In *HPCS*, pages 525–532. IEEE, 2011.

[141] B. Bierbaum, C. Clauss, R. Finocchiaro, M. Pöppe, S. Schuch, and Joachim Worringen. *MP-MPICH – User Documentation and Technical Notes.*, 2009.

[142] T. Mattson and R.van der Wijngaart. Rcce: A small library for many-core communication. Technical report, Jan 2011.

[143] E. Chan. Rcce comm: A collective communication library for the intel single-chip cloud computer. Technical report, 2010.

[144] J. Matienzo and N. D. E. Jerger. Performance analysis of broadcasting algorithms on the intel single-chip cloud computer. In *2012 IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, USA, 21-23 April, 2013*, pages 163–172. IEEE, 2013.

[145] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. High-performance rma-based broadcast on the intel scc. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 121–130, New York, NY, USA, 2012. ACM.

[146] D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper. Asynchronous broadcast on the intel scc using interrupts. In *MARC Symposium*, pages 24–29. ONERA, The French Aerospace Lab, 2012.

[147] H. Al-Khalissi and M. Berekovic. Performance of rcce broadcast algorithm in scc. In *MARC Symposium*, pages 93–98, 2011.

[148] C. Clauss, S. Lankes, P. Reble, J. Galowicz, S. Pickartz, and T. Bemmerl. iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – Version 2.0 iRCCE FLAIR. Technical

report, Chair for Operating Systems, RWTH Aachen University, Kopernikusstr. 16, 52056 Aachen, Germany, March 2013. Users' Guide and API Manual.

[149] J. Nolte, Y. Ishikawa, and M. Sato. Taco: Prototyping high-level object-oriented programming constructs by means of template based programming techniques. *SIGPLAN Not.*, 36(12):35–49, Dec 2001.

[150] T. Prescher, R. Rotta, and J. Notle. Flexible sharing and replication mechanisms for hybrid memory architectures. In *Proceeding of the 4th Many-core Applications Research Community (MARC) Sympisum. Technische Berichte des hasso-Plattner-Instituts für Softwaresystemtechnick an der Universität Potsdam*, pages 67–72, 2012.

[151] Intel Corporation. Pop-shm users guide. Technical report, 2011.

[152] *Software-Managed Cache Coherence for SCC Revision 1.5. [Online]*, 2012.

[153] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. A Coherent and Managed Runtime for ML on the SCC. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 20–25, November 2012.

[154] J. Kim, S. Seo, and J. Lee. An efficient software shared virtual memory for the single-chip cloud computer. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 4:1–4:5, New York, NY, USA, 2011. ACM.

[155] A. Papagiannis and D. S. Nikolopoulos. Scalable runtime support for data intensive applications on the single chip cloud computer. In *Proceedings of the 3rd MARC Symposium, Ettlingen, Germany*, pages 25–30, July 2011.

[156] M. Verstraaten, C. Grelck, M. W. van Tol, R. Bakker, and C. R. Jesshope. Mapping distributed s-net on the 48-core intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany,*, pages 41–46, July 2011.

[157] M. W. van Tol and J. Koivisto. Extending and implementing the self-adaptive virtual processor for distributed memory architectures. *CoRR*, abs/1104.3876, 2011.

[158] A. Prell and T. Rauber. Go's concurrency constructs on the SCC. In *6th Many-core Applications Research Community (MARC) Symposium. Proceedings of the 6th MARC Symposium, 19-20 July 2012, Toulouse, France*, pages 2–6, 2012.

[159] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee. An opencl framework for homogeneous manycores with no hardware cache coherence. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 56–67, Oct 2011.

[160] A. Munshi. The opencl specification version: 1.1 document revision: 44, 2011.

[161] A. Diavastos, G. Stylianou, and P. Trancoso. Tfluxscc: A case study for exploiting performance in future many-core systems. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 26:1–26:2, New York, NY, USA, 2014. ACM.

[162] K. Chapman, A. Hussein, and A. L. Hosking. X10 on the single-chip cloud computer: Porting and preliminary performance. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 7:1–7:8, New York, NY, USA, 2011. ACM.

[163] C. Clauss, S. Pickartz, S. Lankes, and T. Bemmerl. Towards a multicore communications api implementation (mcapi) for the intel single-chip cloud computer (scc). In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 148–155, June 2012.

[164] G. Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. 2010.

[165] V. Gramoli, R. Guerraoui, and V. Trigonakis. Tm2c: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 351–364, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3.

[166] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. Ch. Philos. A microbenchmark study of openmp overheads under nested parallelism. In *IWOMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2008.

[167] H. Al-Khalissi, A. Marongiu, and M. Berekovic. An approach for supporting openmp on the intel scc. In *SPLASH-MARC*, 2013.

[168] A. Marongiu, P. Burgio, and L. Benini. Evaluating openmp support costs on mpsocs. In *DSD*, pages 191–198. IEEE, 2010.

[169] M. Booth and K. Misegades. Microtasking: a new way to harness multiprocessors. In *Cray Channels*, pages 24–27, 1986.

[170] V. Gramoli, R. Guerraoui, and V. Trigonakis. Tm^{2}c: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 351–364, New York, NY, USA, 2012. ACM.

[171] A. Marongiu, P. Burgio, and L. Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *DATE*, pages 105–110, 2012.

[172] Intel® openmp* runtime library, December 2013. URL www.openmprtl.org.

[173] J. Bernard. Using the clone() system call, November 2000. URL http://www.linuxjournal.com/article/5211?page=0,0.

[174] J. He, W. Chen, G. Chen, W. Zheng, Z. Tang, and H. Ye. Openmdsp: Extending openmp to program multi-core dsp. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 288–297. IEEE Computer Society, 2011.

[175] J. Sartori and R. Kumar. Low-overhead, high-speed multi-core barrier synchronization. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 18–34, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11514-4, 978-3-642-11514-1.

[176] D. an Mey, T. Haarman, and W. Koschel. Pushing loop-level parallelization to the limit. Rome, Italy, 2002. Fourth European Workshop on OpenMP (EWOMP 2002).

[177] S. Satoh, K. Kusano, and M. Sato. Compiler optimization techniques for openmp programs. In *Scientific Programming*, pages 9–2, 2001.

[178] H. Al-Khalissi, A. Marongiu, and M. Berekovic. Low-overhead barrier synchronization for openmp-like parallelism on the single-chip cloud computer. In *MARC@RWTH*, pages 26–31, 2012.

[179] H. Al-Khalissi, S. A. A. Shah, and M. Berekovic. An efficient barrier implementation for openmp-like parallelism on the intel scc. In *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '14, pages 76–83, Washington, DC, USA, 2014. IEEE Computer Society.

[180] H. Al-Khalissi, R. Buchty, and M. Berekovic. Efficient barrier synchronization for openmp-like parallelism on the intel scc. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 10–17, Dec 2013.

[181] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[182] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[183] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, 1999.

[184] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *International Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[185] Y. Chen, D. Wang, and W. Zheng. Extended overhead analysis for openmp performance tuning. In *WOMPAT*, volume 2716 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2003.

[186] K. Fürlinger. Openmp application profiling — state of the art and directions for the future. *Procedia Computer Science*, 1(1):2107–2114, 2010. {ICCS} 2010.

[187] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for openmp. *J. Supercomput.*, 23(1):105–128, August 2002.

[188] J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of openmp applications. In *WOMPAT*, volume 2104 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2001.

[189] P. Reble, S. Lankes, F. Zeitz, and T. Bemmerl. Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*, Berkeley, CA, USA, June 2012. Poster Paper.

[190] N. S. Arenstorf and H. F. Jordan. Comparing barrier algorithms. *Parallel Computing*, 12(2):157–170, 1989.

[191] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, April 1987.

[192] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl. A fast inter-kernel communication and synchronization layer for metalsvm. In *MARC Symposium*, pages 19–23, 2011.

[193] D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper. Asynchronous broadcast on the intel scc using interrupts. In *MARC Symposium*, pages 24–29, 2012.

[194] B. D. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. *International Journal of Parallel Programming*, 19(3):225–250, 1990.

[195] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: methodology and performance. *SIGMETRICS Perform. Eval. Rev.*, 27(1):23–34, May 1999. ISSN 0163-5999.

[196] B. Lim. Reactive synchronization algorithms for multiprocessors. Technical report, Cambridge, MA, USA, 1995.

[197] H. Al-Khalissi, M. Berekovic, and A. Marongiu. On the relevance of architectural awareness for efficient fork/join support on cluster-based manycores. In *Proceedings of International Workshop on Manycore Embedded Systems*, MES '14, pages 9:9–9:16, New York, NY, USA, 2014. ACM.

[198] O. Tatebe, M. Sato, and S. Sekiguchi. Impact of openmp optimizations for the mgcg method. In *Proceedings of the Third International Symposium on High Performance Computing*, ISHPC '00, pages 471–481, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-41128-3.

[199] G. Bronevetsky, J. C. Gyllenhaal, and B. R. de Supinski. Clomp: Accurately characterizing openmp application overheads. *International Journal of Parallel Programming*, 37(3):250–265, 2009.

[200] K. Fürlinger and M. Gerndt. Analyzing overheads and scalability characteristics of openmp applications. In *VECPAR*, volume 4395 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2006.

[201] A. Marongiu, P. Burgio, and L. Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 105–110, San Jose, CA, USA, 2012. EDA Consortium.

[202] D. an Mey, D. Schmidl, T. Cramer, and M. Klemm. Openmp programming on intel xeon phi coprocessors: An early performance comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012.

[203] Wikipedia. Memory pool. 2014. URL http://en.wikipedia.org/wiki/Memory_pool.

[204] OpenMP Architecture Review Board. Openmp application program interface v.4.0. 2013. URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[205] P. E. Hadjidoukas and V. V. Dimakopoulos. Nested parallelism in the ompi openmp/c compiler. In *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 662–671. Springer, 2007.

[206] A. E. Eichenberger and K. O'Brien. Experimenting with low-overhead openmp runtime on ibm blue gene/q. *IBM Journal of Research and Development*, 57(1/2): 8, 2013.

[207] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of openmp applications with nested parallelism. In *LCR*, volume 1915 of *Lecture Notes in Computer Science*, pages 100–112. Springer, 2000.

[208] M. Sato. Openmp: Parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In *ISSS*, pages 109–111. IEEE Computer Society, 2002.

[209] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 294–301, New York, NY, USA, 1999. ACM.

[210] M. Ojail, R. David, Y. Lhuillier, and A. Guerre. Artm: a lightweight fork-join framework for many-core embedded systems. In *DATE*, pages 1510–1515, 2013.

[211] M. Süß and C. Leopold. Common mistakes in openmp and how to avoid them. In MatthiasS. Mueller, BarbaraM. Chapman, BronisR. de Supinski, AllenD. Malony, and Michael Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 312–323. Springer Berlin Heidelberg, 2008.

[212] H. Li, S. Tandri, M. Stumm, and K.C. Sevcik. Locality and loop scheduling on numa multiprocessors. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 140–147, Aug 1993.

[213] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in openmp? volume 2716 of *Lecture Notes in Computer Science*, pages 147–159. Springer, 2003.

[214] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran handbook*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, January 1994.

[215] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers, October 1995. URL http://www.cs.virginia.edu/stream.

[216] N. Melot, K. Avdic, C. Kessler, and J. Keller. Investigation of main memory bandwidth on Intel Single-Chip Cloud Computer. Intel MARC3 Symposium 2011, Ettlingen, 2011.

[217] J. D. McCalpin. The stream2 home page, February 2005. URL http://www.cs.virginia.edu/stream/stream2/.

[218] M. Sato, K. Kusano, and S. Satoh. Openmp benchmark using parkbench. In *Proceeding of Second European Workshop on OpenMP, Edinburgh, Scotland, U.K.*, Sept 2000.

[219] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *In Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[220] P. Gschwandtner, T. Fahringer, and R. Prodan. Performance analysis and benchmarking of the intel scc. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 139–149, Washington, DC, USA, 2011. IEEE Computer Society.

[221] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[222] C. Rodriguez and F. de Sande. The openmp source code repository. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244–250, Feb 2005.

[223] Y. Yu and S.T. Acton. Speckle reducing anisotropic diffusion. *Image Processing, IEEE Transactions on*, 11(11):1260–1270, Nov 2002.

[224] The top 500 list. URL http://www.top500.org.

[225] Wikipedia. Mandelbrot set. URL https://en.wikipedia.org/wiki/Mandelbrot_set.

[226] Joseph Robicheaux. 1998. URL http://www.openmp.org/samples/jacobi.f.

[227] Heated_plate: 2d steady state heat equation in a rectangle. URL http://people.sc.fsu.edu/~jburkardt/c_src/heated_plate/heated_plate.html.

[228] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, March 2009.

[229] R. Rezaur. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition, 2013.

[230] Intel. Intel xeon phi coprocessor system software development guide. Nov 2012.

[231] Intel. An overview of programming for intel xeon processors and intel xeon phi coprocessors. Oct 2012.

[232] J. Cownie, A. Duran, M. Klemm, and L. Lin. The parallel universe magazine. 2014. URL https://software.intel.com/sites/default/files/managed/6a/78/parallel_mag_issue18.pdf.